Stable relations and abstract interpretation of higher-order programs

Benoît Montagu, Thomas Jensen

Stable Relations and Abstract Interpretation of Higher-Order Programs

BENOÎT MONTAGU and THOMAS JENSEN, Inria, France

We present a novel denotational semantics for the untyped call-by-value λ-calculus, where terms are interpreted as *stable relations*, *i.e.* as binary relations between substitutions and values, enjoying a monotonicity property. The denotation captures the input-output behaviour of higher-order programs, and is proved sound and complete with respect to the operational semantics. The definition also admits a presentation as a program logic. Following the principles of abstract interpretation, we use our denotational semantics as a *collecting semantics* to derive a modular *relational* analysis for higher-order programs. The analysis infers equalities between the arguments of a program and its result—a form of *frame condition* for functional programs.

CCS Concepts: • **Theory of computation** → **Program analysis**; **Abstraction**; *Denotational semantics*; *Functional constructs*.

Additional Key Words and Phrases: static analysis, λ-calculus, abstract interpretation, correlations

ACM Reference Format:
Benoît Montagu and Thomas Jensen. 2020. Stable Relations and Abstract Interpretation of Higher-Order Programs. *Proc. ACM Program. Lang.* 4, ICFP, Article 119 (August 2020), 30 pages. https://doi.org/10.1145/3409001

1 INTRODUCTION

Finding an upper bound of the effect that a program can have on its environment is a central problem in semantics and program verification. For instance, *frame conditions* [Meyer 2015] specify which parts of an object or which global variables might be modified by a program. They are an essential ingredient of tools for deductive program verification [Barnett et al. 2006, 2005; Filliâtre and Paskevich 2013; Marché and Paulin-Mohring 2005]. *Framing* is also a central notion of separation logics [Reynolds 2002], that leverages *local reasoning* to verify imperative programs more concisely. Purely functional programs, however, have *no* effect on their environment. A notion of *frame condition for functional programs* would be to determine which parts of the inputs of a program agree with which parts of its output. More generally, this amounts to finding a precise *relation* between the inputs of a program and its output.

Functional programs are involved in the specification and the verification of large programs. Examples include compilers [Kumar et al. 2014; Leroy 2006] or operating systems [Gu et al. 2011; Klein et al. 2009]. Such endeavours are labour-intensive, and could benefit from the automatic inference of relations between inputs and outputs. A recent experiment [Andreescu et al. 2019] indeed showed that, by inferring such relations, almost two thirds of the proof obligations for verifying a micro-kernel could be *automatically* discharged. Optimising compilers for functional languages could benefit, too, from such analyses, as compilers could detect which values are unchanged across function calls, and perform more aggressive eliminations of redundant expressions.

Authors' address: Benoît Montagu, benoit.montagu@inria.fr; Thomas Jensen, thomas.jensen@inria.fr, Inria, Campus universitaire de Beaulieu, Avenue du Général Leclerc, Rennes, 35042, France.

2475-1421/2020/8-ART119
https://doi.org/10.1145/3409001

The goal of this paper is to find sound approximations of input-output relations for *higher-order* functional programs. For example, we would like to detect that the result $y = f\,x$ of calling a function f on some argument x is a structure that satisfies $y.\mathrm{a.c} = \mathrm{Some}(x.\mathrm{b})$. The higher-order aspect is of paramount importance, as it opens up the possibility to analyse the input-output behaviour of *monadic code* and of generic libraries. The higher-order setting also provides incentive to design *modular* analyses [Cousot and Cousot 2002], that are more likely to scale to large code bases. To achieve this goal, we rely on the theory and methodology of abstract interpretation [Cousot and Cousot 1977].

This methodology advises to first define a *collecting semantics*—that expresses the most *precise* properties one wishes to infer—and then derive a static analysis, by means of successive abstractions of the collecting semantics. Cachera and Pichardie [2010] closely follow this approach for a WHILE language. They first define a denotational interpreter, that computes *concrete* properties, which they formally prove sound with respect to an operational semantics. Then, they apply abstractions to their interpreter with little difficulty, because most of the complexity lies in the soundness proof of the interpreter. We follow the same methodology: we first define a denotational semantics of a higher-order language, that we prove *sound and complete*, and then obtain a static analysis in a second stage, by applying several abstractions. The first semantic stage factors out the main difficulties, and alleviates the abstraction stage.

Previous work on static analysis of higher-order languages defined collecting semantics as *sets* of values (or of final configurations for some abstract machine), obtained from running a program on a set of initial inputs. In contrast, we define a novel collecting semantics, that builds *relations* between the unknown inputs of a program and its output values. We use this semantics as an intermediate step to derive a relational, modular analysis for a higher-order language. More generally, we think our semantics constitutes a *promising* starting point for the design of new relational analyses for higher-order functional programs. It could also help increase the precision of existing analyses of higher-order programs—such as control flow analyses—by exploiting relational information.

In this paper, we present the following contributions:

- We define a denotational-style *collecting semantics* for the untyped call-by-value λ-calculus (§2), that interprets programs as binary relations between substitutions (defining input values) and output values, that enjoy a monotonicity property. We call them *stable* relations. A specificity of our approach is that the denotations might involve the *names* of some inputs.
- Because of the presence of names, it becomes necessary to ensure that the definitions are well-behaved with respect to renamings. We solve the problem of properly handling these names (§3) by leveraging nominal techniques [Gabbay and Pitts 1999; Pitts 2016].
- We prove the *soundness and completeness* of our semantics with respect to the operational semantics (§4). We also offer an equivalent presentation of our semantics in the form of a program logic, that could be used to derive relations between inputs and outputs of programs.
- We demonstrate the relevance of the collecting semantics by deriving, by means of successive abstractions, a static analyser for simply typed λ-terms that infers relations between inputs and outputs (§5), thereby computing *frame conditions* for functional programs. We implemented a prototype analyser in OCaml that closely follows the formal development. We discuss the precision of the analysis on example programs (§5.3).
- Our analysis uses the abstract domain of correlations [Andreescu et al. 2019] to represent binary relations over values. Guided by the abstractions of (§5.1), we extend the correlation abstract domain from first-order to higher-order values (§5.2).
- We verified in the Coq proof assistant the formal properties presented in this paper (§6). The Coq development is available as an accompanying artefact.

$$\frac{}{(\lambda \mathsf{x}.\, t)\, v \rightsquigarrow t\,[\mathsf{x} \leftarrow v]} \qquad \frac{}{\mathsf{let}\,\mathsf{x} = v\,\mathsf{in}\,t \rightsquigarrow t\,[\mathsf{x} \leftarrow v]} \qquad \frac{t \rightsquigarrow u}{t\,t' \rightsquigarrow u\,t'} \qquad \frac{t \rightsquigarrow u}{v\,t \rightsquigarrow v\,u}$$

$$\frac{i \in \{1,2\}}{\pi_i(v_1, v_2) \rightsquigarrow v_i} \qquad \frac{t \rightsquigarrow u}{(t, t') \rightsquigarrow (u, t')} \qquad \frac{t \rightsquigarrow u}{(v, t) \rightsquigarrow (v, u)} \qquad \frac{t \rightsquigarrow u}{\pi_i\, t \rightsquigarrow \pi_i\, u} \qquad \frac{t \rightsquigarrow u}{\mathsf{inj}_i\, t \rightsquigarrow \mathsf{inj}_i\, u}$$

$$\frac{t \rightsquigarrow u}{\begin{array}{ll}\mathsf{match}\ t\ \mathsf{with} & \mathsf{match}\ u\ \mathsf{with}\\ \mid \mathsf{inj}_1\,\mathsf{x}_1 \rightarrow t_1 & \rightsquigarrow \mid \mathsf{inj}_1\,\mathsf{x}_1 \rightarrow t_1\\ \mid \mathsf{inj}_2\,\mathsf{x}_2 \rightarrow t_2 & \mid \mathsf{inj}_2\,\mathsf{x}_2 \rightarrow t_2\end{array}} \qquad \frac{i \in \{1,2\}}{\begin{array}{ll}\mathsf{match}\ \mathsf{inj}_i\, v\ \mathsf{with} & \\ \mid \mathsf{inj}_1\,\mathsf{x}_1 \rightarrow t_1 & \rightsquigarrow t_i\,[\mathsf{x}_i \leftarrow v]\\ \mid \mathsf{inj}_2\,\mathsf{x}_2 \rightarrow t_2 & \end{array}}$$

Fig. 1. Small-step semantics

2 INTERPRETATION OF λ-TERMS AS STABLE RELATIONS

2.1 Preliminary Definitions: The Untyped λ-Calculus

We consider the *untyped* λ-calculus equipped with a standard call-by-value reduction semantics. This language features pairs, a unit value, and binary sums.

Definition 2.1 (Syntax). The syntax of terms is inductively defined as follows:

$$\begin{array}{rcl} t, u & ::= & \mathsf{x} \mid \mathsf{let}\,\mathsf{x} = t\,\mathsf{in}\,u \mid \lambda \mathsf{x}.\, t \mid t\,u \mid () \mid (t, u) \mid \pi_1\, t \mid \pi_2\, t \\ & \mid & \mathsf{inj}_1\, t \mid \mathsf{inj}_2\, t \mid \mathsf{match}\ t\ \mathsf{with}\ \mathsf{inj}_1\,\mathsf{x}_1 \rightarrow u_1 \mid \mathsf{inj}_2\,\mathsf{x}_2 \rightarrow u_2 \end{array}$$

We consider terms up to α-equivalence, and write $\mathrm{fv}\, t$ for the set of free variables of the term t. A term t is *closed* when $\mathrm{fv}\, t = \emptyset$. The values of the language are standard in a weak eager semantics.

Definition 2.2 (Values). Values are inductively defined as follows:

$$v \quad ::= \quad \lambda \mathsf{x}.\, t \mid () \mid (v, v) \mid \mathsf{inj}_1\, v \mid \mathsf{inj}_2\, v$$

Values can contain free variables. We write $\mathcal{V}$ to denote the set of *closed* values.

We write $t\,[\mathsf{x} \leftarrow u]$ to denote the term t in which the variable x is substituted by the term u. We recall in Fig. 1 the call-by-value small-step semantics. We write $t \rightsquigarrow u$ to denote that the term t *reduces* to the term u in *one* step. We write $t \rightsquigarrow^\star u$ for its reflexive transitive closure. The language is untyped; fixpoint combinators can thus be defined within the language.

2.2 Substitutions and Stable Relations

We are interested in modelling the input-output behaviour of programs. A first question one should ask is: *what are the inputs of a λ-term?* To answer this question, it is easier to consider *open* terms: the inputs of an open term are the unknown values it depends on, that is to say, the possible valuations for its free variables. Providing values to free variables is the purpose of substitutions. Substitutions $\sigma \in \Sigma_v$ are finite maps from variables to closed values. Applying a substitution σ to a term t, written $t \cdot \sigma$, consists in replacing every free variable x of t with $\sigma(\mathsf{x})$. Our goal of modelling the input-output behaviour of t is therefore the following: find relations between the substitutions σ that close t and the normal form of the closed term $t \cdot \sigma$, should this normal form exist. In the case of closed terms, this goal boils down to finding the possible normal forms.

Given a set of value substitutions $\mathcal{I} \subseteq \Sigma_v$, we define the *collecting semantics* of a term t as the set $\langle\!|t|\!\rangle_{\mathcal{I}} \triangleq \bigcup_{\sigma \in \mathcal{I}} \{(\sigma, v) \mid t \cdot \sigma \rightsquigarrow^\star v\}$. It computes the most precise relation between some inputs and the possible output values. This definition contrasts with the definition of reachable values

$\bigcup_{\sigma \in \mathcal{I}} \{v \mid t \cdot \sigma \rightsquigarrow^\star v\}$, where the link between inputs and outputs is forgotten. Our definition of $(\!|t|\!)_{\mathcal{I}}$ keeps this link, which is essential to derive a *relational* analysis of programs.

When two substitutions σ_1 and σ_2 agree on the values assigned to the free variables of t, their applications on t give the same results. A special case is when σ_2 is an extension of σ_1, with new bindings for the variables not recorded in σ_1. We write $\sigma_1 \sqsubseteq \sigma_2$ in that case, and say that σ_2 *extends* σ_1. The formal definition follows.

Definition 2.3 (Extension ordering on substitutions). We say that σ_2 extends σ_1, written $\sigma_1 \sqsubseteq \sigma_2$ when $\operatorname{dom}\sigma_1 \subseteq \operatorname{dom}\sigma_2$, and for every $\mathsf{x} \in \operatorname{dom}\sigma_1$, $\sigma_1(\mathsf{x}) = \sigma_2(\mathsf{x})$.

Lemma 2.4 (Preorder). *The extension ordering is a* preorder*: It is a reflexive, transitive relation.*

The fact that $t \cdot \sigma_1 = t \cdot \sigma_2$ when $\sigma_1 \sqsubseteq \sigma_2$ and $\operatorname{fv} t \subseteq \operatorname{dom}\sigma_1$ indicates that the set $\mathcal{I}$ can be closed upwards using $\sqsubseteq$ without changing the output values in $(\!|t|\!)_{\mathcal{I}}$. When $\mathcal{I}$ is upward closed, the relation $(\!|t|\!)_{\mathcal{I}}$ is such that whenever $(\sigma_1, v) \in (\!|t|\!)_{\mathcal{I}}$ and $\sigma_1 \sqsubseteq \sigma_2$, we also have $(\sigma_2, v) \in (\!|t|\!)_{\mathcal{I}}$. We call this property on relations *stability*.

Definition 2.5 (Stable relation). A relation $\mathcal{R} \in \wp(\Sigma_v \times \mathcal{V})$ is *stable* when for every substitutions σ_1 and σ_2, and for every value v, if $(\sigma_1, v) \in \mathcal{R}$ and $\sigma_1 \sqsubseteq \sigma_2$, then $(\sigma_2, v) \in \mathcal{R}$.

By exploiting the isomorphism between $\wp(\Sigma_v \times \mathcal{V})$ and $\Sigma_v \to \wp(\mathcal{V})$, we can restate stability: a stable relation is (up to isomorphism) a monotone function from $(\Sigma_v, \sqsubseteq)$ to $(\wp(\mathcal{V}), \subseteq)$. What we call *stability* is akin to the notion of *persistence* in Kripke semantics: a proposition P is persistent when the truth of P in some world w implies the truth of P in any world w' that is larger than w.

A first contribution of this paper is to define an interpretation of λ-terms as *stable* relations between substitutions and closed values. That interpretation is sound and complete with respect to the operational semantics. In the rest of the section we guide the reader by giving intuitions on some interpretation rules, as a means to understand how and why our construction is sound. We will introduce definitions as needed. We have taken some combinators on relations from the relators of allegories [Bird and de Moor 1996] and from the abstract domain of correlations [Andreescu et al. 2019]. Some other combinators are novel. They are all defined in Fig. 2.

In the rest of the section, we write $E \vdash t : \mathcal{R}$ to denote that in the environment E, the input-output behaviour of the term t is over-approximated by the relation $\mathcal{R}$. An environment E maps some variables to relations. We postpone the formal definition of environments and instead focus on motivating and giving the reader some intuition first. The environment denotes constraints on the admissible inputs, *i.e.* it denotes a set of substitutions that are considered as possible inputs for the evaluation of the term t. We write this set of substitutions $[\![E]\!]$. As a first approximation, the reader can assume that if $\sigma \in [\![E]\!]$, then $\operatorname{dom}\sigma = \operatorname{dom} E$.

We postpone to §4 the formal definition of the interpretation, its proof of soundness and completeness, and the presentation of an equivalent version in the form of a program logic.

2.3 Interpretation of Products

We begin with the interpretation of the constructs related to products. The interpretation of these constructs is straightforward, but illustrates well how the semantics relates substitutions and values.

$$\frac{E \vdash t : \mathcal{R}}{E \vdash \pi_1\, t : \mathcal{R}; \mathsf{PAIR}^{\mathsf{L}}(\mathsf{EQ}, \top_{\mathsf{v}})} \qquad \frac{E \vdash t : \mathcal{R}}{E \vdash \pi_2\, t : \mathcal{R}; \mathsf{PAIR}^{\mathsf{L}}(\top_{\mathsf{v}}, \mathsf{EQ})}$$

To interpret $\pi_1\, t$, we first interpret t as a relation $\mathcal{R}$. This relation relates a substitution to the normal form of t. This normal form might be, or not, a pair. If it is not a pair, the evaluation should fail, *i.e.* return the empty relation. If the normal form is a pair, one should retrieve its first component to build

a relation between a substitution and that first component. To perform this operation, we compose $\mathcal{R}$ on the right-hand side with the relation $\mathsf{PAIR}^{\mathsf{L}}(\mathsf{EQ}, \top_{\mathsf{v}})$, that performs the actual projection on pairs. The composition is the usual composition on relations: $\mathcal{R}_1; \mathcal{R}_2 \triangleq \{(a, c) \mid \exists b, (a, b) \in \mathcal{R}_1 \wedge (b, c) \in \mathcal{R}_2\}$. The relation $\mathsf{PAIR}^{\mathsf{L}}(\mathsf{EQ}, \top_{\mathsf{v}})$ is equivalent to $\{((v_1, v_2), v_1) \mid v_1, v_2 \in \mathcal{V}\}$, that indeed extracts the first component of a pair. That relation is built from the basic relations EQ and $\top_{\mathsf{v}}$ and from a combinator $\mathsf{PAIR}^{\mathsf{L}}(\mathcal{R}_1, \mathcal{R}_2)$. The equality (or identity) relation EQ is the relation $\{(v, v) \mid v \in \mathcal{V}\}$, whereas $\top_{\mathsf{v}} \triangleq \mathcal{V} \times \mathcal{V}$. The relation $\mathsf{PAIR}^{\mathsf{L}}(\mathcal{R}_1, \mathcal{R}_2) \triangleq \{((v_1, v_2), v) \mid (v_1, v) \in \mathcal{R}_1 \wedge (v_2, v) \in \mathcal{R}_2\}$ relates a pair (v_1, v_2) on the left-hand side to some other value v on the right-hand side, such that the first component v_1 is related to v using $\mathcal{R}_1$ and the second component v_2 is related to v using $\mathcal{R}_2$. The superscript L indicates that the pair is on the left-hand side of the relation. There is a dual superscript R that will appear for the rule for pair creation below. The rule to evaluate $\pi_2\, t$ is similar to that of $\pi_1\, t$: to retrieve the second component of the pair, we compose on the right-hand side with $\mathsf{PAIR}^{\mathsf{L}}(\top_{\mathsf{v}}, \mathsf{EQ})$.

The rule to interpret the creation of a pair (t_1, t_2)

$$\frac{E \vdash t_1 : \mathcal{R}_1 \qquad E \vdash t_2 : \mathcal{R}_2}{E \vdash (t_1, t_2) : \mathsf{PAIR}^{\mathsf{R}}(\mathcal{R}_1, \mathcal{R}_2)}$$

evaluates each t_i into some relation $\mathcal{R}_i$, which relates a substitution with the normal form of t_i. The resulting relation must relate a substitution with the normal form of (t_1, t_2), *i.e.* with the pair of the normal forms of t_1 and t_2. We create such a relation using the combinator $\mathsf{PAIR}^{\mathsf{R}}(\mathcal{R}_1, \mathcal{R}_2) \triangleq \{(\sigma, (v_1, v_2)) \mid (\sigma, v_1) \in \mathcal{R}_1 \wedge (\sigma, v_2) \in \mathcal{R}_2\}$, that relates a substitution σ on the left-hand side with a pair of values (v_1, v_2) on the right-hand side, such that σ is related to the first component v_1 using $\mathcal{R}_1$, and σ is related to the second component v_2 using $\mathcal{R}_2$. We write R as a superscript to indicate that the pair should be on the right-hand side of the relation.

2.4 Interpretation of let-Bindings

The rule for let-bindings illustrates how we handle environments and substitutions.

$$\frac{E \vdash t_1 : \mathcal{R}_1 \qquad E, \mathsf{x} : \mathcal{R}_1 \vdash t_2 : \mathcal{R}_2 \qquad x \notin \operatorname{dom} E}{E \vdash \mathsf{let}\ \mathsf{x} = t_1\ \mathsf{in}\ t_2 : \mathsf{LET}^{\operatorname{dom} E}\ \mathsf{x} \leftarrow \mathcal{R}_1\ \mathsf{IN}\ \mathcal{R}_2}$$

To interpret $\mathsf{let}\ \mathsf{x} = t_1\ \mathsf{in}\ t_2$, one first evaluates t_1 into a relation $\mathcal{R}_1$, and then evaluates t_2 in the environment extended with a binding $x : \mathcal{R}_1$. This gives us a relation $\mathcal{R}_2$. Before we start explaining how we build the resulting relation, we remark two important things.

First, this rule tells us what the environment E should hold. The relation $\mathcal{R}_1$ relates a substitution σ with some value v_1, where σ is a substitution whose domain is $\operatorname{dom} E$, and that satisfies the hypotheses of the environment E. In the second premise, when we extend the environment E with that relation, we specify for the interpretation of t_2 a set of valid substitutions whose domain is $\{\mathsf{x}\} \cup \operatorname{dom} E$: the substitutions are of the form $\sigma\,[\mathsf{x} \mapsto v_1]$ such that σ satisfies the constraints of E, and such that $(\sigma, v_1) \in \mathcal{R}_1$. This is exactly what we obtain from the interpretation of t_1. The first element we just learned is that the set of substitutions represented by E, written $[\![E]\!]$, is such that if $\sigma \in [\![E]\!]$ and $(\sigma, v) \in \mathcal{R}$, then $\sigma\,[\mathsf{x} \mapsto v] \in [\![E, \mathsf{x} : \mathcal{R}]\!]$. This remark highlights the *dependencies* that exist between the different bindings of an environment. Consider for example the environment $E_0 = \mathsf{x} : \top, \mathsf{y} : \mathcal{R}$, where $\mathcal{R} = \{(\sigma, v) \mid v = (\sigma(\mathsf{x}), \mathsf{inj}_1\,())\}$. The relation $\mathcal{R}$ that is associated to the variable y depends on the previous binding x. This environment E_0 imposes that the substitutions σ that inhabit E_0 satisfy a constraint between $\sigma(\mathsf{x})$ and $\sigma(\mathsf{y})$, namely: it must be true that $\sigma(\mathsf{y}) = (\sigma(\mathsf{x}), \mathsf{inj}_1\,())$. Similar dependencies are found in typing environments for

dependent types. Indeed, the meaning of a binding $\mathsf{x} : \mathcal{R}$ depends on the meaning of E, that is the environment that lies *before* $\mathsf{x} : \mathcal{R}$. We defer to §4 the formal definition of $[\![E]\!]$.

The second remark about the interpretation rule for let-bindings is that the relation $\mathcal{R}_2$ relates a substitution σ_2 with the normal form of t_2, and σ_2 must have $\mathrm{dom}(E, \mathsf{x} : \mathcal{R}_1)$ as domain. In particular, it must contain a binding for x. However, the final relation that we expect from the evaluation of the let-expression must relate a substitution σ' with the result of t_2, where $\mathrm{dom}\,\sigma' = \mathrm{dom}\,E$. Thus, we cannot directly return $\mathcal{R}_2$, because its substitutions have one extra binding for x. One should somehow get rid of this extra binding. The next paragraphs present a solution for binding removal.

Which relation should one return to interpret let-expressions? We first give a slightly incorrect answer, so as to convey most intuitions to the reader. Intuitively, the relation should relate σ and v_2 such that σ is related to some v_1 using $\mathcal{R}_1$, and such that $\sigma[\mathsf{x} \mapsto v_1]$ is related to v_2 using $\mathcal{R}_2$. In other words, one should return $\{(\sigma, v_2) \mid \exists v_1, (\sigma, v_1) \in \mathcal{R}_1 \wedge (\sigma[\mathsf{x} \mapsto v_1], v_2) \in \mathcal{R}_2\}$. Let us introduce the notation $\mathcal{R}[\mathsf{x} \mapsto v] \triangleq \{(\sigma, \sigma[\mathsf{x} \mapsto v]) \mid (\sigma, v) \in \mathcal{R}\}$, that relates a substitution with the same substitution extended with a binding $[\mathsf{x} \mapsto v]$ whose value v belongs to $\mathcal{R}$. Using this notation, our relation rewrites to $\bigcup_{v_1} \mathcal{R}_1[\mathsf{x} \mapsto v_1]\,;\mathcal{R}_2$. It reads as follows: take any value from the output of $\mathcal{R}_1$ and add it to the input substitution, and then give it as input to $\mathcal{R}_2$. This closely matches the big-step evaluation rule for let-expressions in natural semantics [Kahn 1987]!

There is a significant semantic subtlety, though: how should we deal with the variable x? The variable x is indeed a *bound* variable of the let-expression, and any sufficiently fresh variable could be chosen as well: the choice of that variable name should not matter in the relation that our rule outputs. In other words, the variable x should behave as a bound variable in our final relation too, so that the *name* of the local variable x cannot escape and be mistakenly considered as a global input. We need a notion of free variable and of α-equivalence for our relations. Defining these notions would have been easy, had we chosen a *syntax* for the elements of our domain of interpretation. Instead, we have chosen to stay at a semantic level, and have chosen relations, which are semantic objects. The right tool to solve this issue comes from nominal techniques [Gabbay and Pitts 1999; Pitts 2016]. Based on nominal techniques, we can define an operation $[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot \mathcal{R}$ on relations that exchanges the variables x for y (and conversely) in the relation $\mathcal{R}$. Then, we can define $\exists^{\mathcal{S}}[\mathsf{x}].\mathcal{R} \triangleq \bigcup_{y \notin \mathcal{S}} [y \leftrightarrow \mathsf{x}] \cdot \mathcal{R}$ that closes the relation $\mathcal{R}$ under all possible replacements for the variable x with a variable that avoids a finite set $\mathcal{S}$. This technical device effectively makes x behave as a bound variable, by allowing any choice for x outside the set $\mathcal{S}$. The reader can ignore this technical issue in a first read. We give more details on how we deal with nominal issues in §3.

We are now ready to give the final result of the interpretation for let-expressions: $\mathsf{LET}^{\mathrm{dom}\,E}\ \mathsf{x} \leftarrow \mathcal{R}_1\ \mathsf{IN}\ \mathcal{R}_2 \triangleq \exists^{\mathrm{dom}\,E}[\mathsf{x}].\bigcup_{v_1} \mathcal{R}_1[\mathsf{x} \mapsto v_1]\,;\mathcal{R}_2$. It reads as follows: choose a variable x that is fresh for $\mathrm{dom}\,E$, then find a value in the output of $\mathcal{R}_1$ (this is the interpretation of t_1), extend the substitution with that value, and give it as an input to $\mathcal{R}_2$ (this is the interpretation of t_2).

2.5 Interpretation of Sums

We are now ready to explain the interpretation for sums.

$$\frac{E \vdash t : \mathcal{R}}{E \vdash \mathsf{inj}_1\, t : \mathsf{SUM}^{\mathsf{R}}(\mathcal{R}, \bot)} \qquad \frac{E \vdash t : \mathcal{R}}{E \vdash \mathsf{inj}_2\, t : \mathsf{SUM}^{\mathsf{R}}(\bot, \mathcal{R})}$$

The interpretation rules are similar to the one for the pair constructor. To interpret a left injection $\mathsf{inj}_1\, t$, one first interprets t into a relation $\mathcal{R}$, that relates a substitution to the normal form v of t. The expected final relation must however relate a substitution to the left injection of v. We return the relation $\mathsf{SUM}^{\mathsf{R}}(\mathcal{R}, \bot)$ that is equivalent to $\{(\sigma, \mathsf{inj}_1\, v) \mid (\sigma, v) \in \mathcal{R}\}$, that builds the desired final relation. The combinator $\mathsf{SUM}^{\mathsf{R}}(\mathcal{R}_1, \mathcal{R}_2) \triangleq \{(\sigma, \mathsf{inj}_1\, v_1) \mid (\sigma, v_1) \in \mathcal{R}_1\} \cup \{(\sigma, \mathsf{inj}_2\, v_2) \mid (\sigma, v_2) \in \mathcal{R}_2\}$

relates a substitution on the left-hand side with a sum value in the right-hand side, such that $\mathcal{R}_1$ is used to relate the underlying value in case of a left injection, and $\mathcal{R}_2$ is used in case of a right injection. We use R as a superscript to indicate that the sum is on the right-hand side of the relation. The occurrence of $\bot$ in $\mathsf{SUM}^{\mathsf{R}}(\mathcal{R}, \bot)$ asserts that no right injection can be produced.

The interpretation of right injection is similar to the one of the left injection, but uses the combinator $\mathsf{SUM}^{\mathsf{R}}(\bot, \mathcal{R})$ instead, so as to create a right injection value.

The interpretation of sum elimination requires a bit more explanation.

$$\frac{\begin{array}{c} E \vdash t : \mathcal{R} \qquad \mathsf{x}_1 \notin \operatorname{dom} E \qquad \mathsf{x}_2 \notin \operatorname{dom} E \\ E, \mathsf{x}_1 : \mathcal{R}; \mathsf{SUM}^{\mathsf{L}}(\mathsf{EQ}, \bot) \vdash t_1 : \mathcal{R}_1 \qquad E, \mathsf{x}_2 : \mathcal{R}; \mathsf{SUM}^{\mathsf{L}}(\bot, \mathsf{EQ}) \vdash t_2 : \mathcal{R}_2 \end{array}}{E \vdash \mathsf{match}\ t\ \mathsf{with}\ \mathsf{inj}_1\, \mathsf{x}_1 \to t_1 \mid \mathsf{inj}_2\, \mathsf{x}_2 \to t_2 : \mathsf{MATCH}^{\operatorname{dom} E}\ \mathcal{R}\ \mathsf{WITH}\ \mathsf{x}_1 \leftarrow \mathcal{R}_1 \mid \mathsf{x}_2 \leftarrow \mathcal{R}_2}$$

The expression match t with $\mathsf{inj}_1\, \mathsf{x}_1 \to t_1 \mid \mathsf{inj}_2\, \mathsf{x}_2 \to t_2$ is interpreted as follows. First, t is interpreted into a relation R. Then, t_1 is interpreted into a relation $\mathcal{R}_1$ in a context that is extended with the binding $\mathsf{x}_1 : \mathcal{R}; \mathsf{SUM}^{\mathsf{L}}(\mathsf{EQ}, \bot)$. This new binding tells us that the normal form of t is necessarily of the form $\mathsf{inj}_1\, v_1$ (the case $\mathsf{inj}_2\, v_2$ is rendered impossible by the presence of $\bot$ in the left-sum combinator), and that x_1 is bound to the value v_1 (as indicated by EQ). The case for t_2 is similar, and gives us $\mathcal{R}_2$ under the assumption that t leads to a right injection. Thus, the evaluations of the two branches t_1 and t_2 record in their environment a refined information about the value of t.

Then, similarly to the case of let-expressions, we eliminate the local variables x_1 and x_2: in the first branch, we get $\mathsf{LET}^{\operatorname{dom} E}\ \mathsf{x}_1 \leftarrow \mathcal{R}; \mathsf{SUM}^{\mathsf{L}}(\mathsf{EQ}, \bot)\ \mathsf{IN}\ \mathcal{R}_1$ and in the second branch $\mathsf{LET}^{\operatorname{dom} E}\ \mathsf{x}_2 \leftarrow \mathcal{R}; \mathsf{SUM}^{\mathsf{L}}(\bot, \mathsf{EQ})\ \mathsf{IN}\ \mathcal{R}_2$. Finally, we obtain as a final relation the union of the two branches: $\mathsf{MATCH}^{\operatorname{dom} E}\ \mathcal{R}\ \mathsf{WITH}\ \mathsf{x}_1 \leftarrow \mathcal{R}_1 \mid \mathsf{x}_2 \leftarrow \mathcal{R}_2 \triangleq (\mathsf{LET}^{\operatorname{dom} E}\ \mathsf{x}_1 \leftarrow \mathcal{R}; \mathsf{SUM}^{\mathsf{L}}(\mathsf{EQ}, \bot)\ \mathsf{IN}\ \mathcal{R}_1) \cup (\mathsf{LET}^{\operatorname{dom} E}\ \mathsf{x}_2 \leftarrow \mathcal{R}; \mathsf{SUM}^{\mathsf{L}}(\bot, \mathsf{EQ})\ \mathsf{IN}\ \mathcal{R}_2)$. Because we use a let-binding at the level of relations for each branch, we ensure that the variables x_1 and x_2 are indeed bound in the final relation.

2.6 Interpretation of Variables, Selfification and Gathering

$$\frac{}{E \vdash \mathsf{x} : E(\mathsf{x})}\ \textsc{Var} \qquad\qquad \frac{}{E \vdash \mathsf{x} : \mathsf{GATHER}(E) \cap \mathsf{SELF}(\mathsf{x})}\ \textsc{Var'}$$

We informally learned from the rule for let bindings, that when we have an environment of the form $E_1, \mathsf{x} : \mathcal{R}, E_2$, the relation $\mathcal{R}$ relates substitutions that define the variables of $\operatorname{dom} E_1$ with some value. To prove the rule of variables correct, we need to build a relation whose substitution on its left-hand side has $\operatorname{dom}(E_1, \mathsf{x} : \mathcal{R}, E_2)$ as a domain, which is larger than $\operatorname{dom} E_1$. Some weakening result is thus needed to transform $\mathcal{R}$ so that it accepts substitutions with larger domains. This is where stable relations come into play: since we only consider stable relations, $\mathcal{R}$ is stable, and therefore closed by extensions of substitutions. If $(\sigma, v) \in \mathcal{R}$, then $\operatorname{dom} \sigma = \operatorname{dom} E_1$ and we can find σ' that is an extension of σ and whose domain is $\operatorname{dom}(E_1, \mathsf{x} : \mathcal{R}, E_2)$. Then we have $(\sigma', v) \in \mathcal{R}$ by stability. The relation $\mathcal{R}$ is therefore a correct over-approximation of the input-output behaviour of x under the hypotheses about inputs, that are expressed by the environment. Thus, the rule Var is sound. Stability is the key ingredient that makes that weakening process possible. By restricting our study to stable relations, we have *internalised* the weakening property.

The expected rule for variables is correct. Rule Var' is correct too, and also more precise. In §4, we show it is *sound and complete*. Var' uses two additional operations: $\mathsf{GATHER}(E)$ and $\mathsf{SELF}(\mathsf{x})$.

The relation $\mathsf{SELF}(\mathsf{x})$ is an instance of the more general form $\mathsf{SELF}(t)$, called *selfification*. It is defined as $\{(\sigma, v) \mid v = t \cdot \sigma \wedge \operatorname{fv} t \subseteq \operatorname{dom} \sigma \wedge \sigma \in \Sigma_v\}$. When specialised with $t = \mathsf{x}$, the definition reduces to $\{(\sigma, \sigma(\mathsf{x})) \mid \sigma \in \Sigma_v\}$. Therefore, $\mathsf{SELF}(\mathsf{x})$ contains the only information that the resulting value was recorded in the variable x in the input substitution, and nothing else. Given the input, $\mathsf{SELF}(\mathsf{x})$ entirely specifies the output value. The $\mathsf{SELF}(\mathsf{x})$ predicate reminds of *strengthening*, found in ML modules [Dreyer et al. 2003; Leroy 2000] and singleton kinds [Stone and Harper 2006].

The relation GATHER(E) constructs a relation that gathers all the constraints on substitutions that are imposed by E. We are only constraining the substitutions here : the values on the right-hand side of the relation GATHER(E) are not constrained. GATHER(E) is defined as $\bigcap_{\mathsf{x}\in\operatorname{dom} E}\{(\sigma, v) \mid \mathsf{x} \in \operatorname{dom}\sigma \wedge (\sigma, \sigma(\mathsf{x})) \in E(\mathsf{x})\}$. It precisely specifies the input substitutions that are accepted by E. The next lemma helps to understand the definition of GATHER(E).

Lemma 2.6. *The relation* GATHER(E) *enjoys an equivalent characterisation by induction on E:*

- *If E is the empty environment, then* $\mathsf{GATHER}(E) = \top$.
- *If* $\mathsf{x} \notin \operatorname{dom} E$ *and* $\mathcal{R} \in \wp(\Sigma_v \times \mathcal{V})$, *then* $\mathsf{GATHER}(E, \mathsf{x} : \mathcal{R}) = \mathsf{GATHER}(E) \cap ((\mathcal{R} \cap \mathsf{SELF}(\mathsf{x})) ; \top_{\mathsf{v}})$.

The relations GATHER(E) and SELF(x) are both sound over-approximations of the input-output behaviour of the program x. Thus, their intersection is also sound, *i.e.* the rule Var' is correct. Rule Var' is more precise than Var: for example, with $E = \mathsf{x} : \top, \mathsf{y} : \bot$, we have $E(\mathsf{x}) = \top$, which is less precise than SELF(x) or than $\mathsf{GATHER}(E) = \bot$. This fact is formally stated by the next lemma.

Lemma 2.7. *Let E be an environment of relations in $\wp(\Sigma_v \times \mathcal{V})$, with no duplicate bindings. If* $\mathsf{x} \in \operatorname{dom} E$, *then* $\mathsf{GATHER}(E) \cap \mathsf{SELF}(\mathsf{x}) \subseteq E(\mathsf{x})$.

The intuition for why Var' is complete is that its two constituents GATHER(E) and SELF(x) entirely specify the left-hand side, and respectively the right-hand side of the relation.

2.7 Interpretation of Functions

We interpret functions, too, as stable relations. We want to obtain a rule similar to a typing rule:

$$\frac{E, \mathsf{x} : \mathcal{R}_1 \vdash t : \mathcal{R}_2 \qquad \mathsf{x} \notin \operatorname{dom} E \qquad \mathrm{wf}_{\operatorname{dom} E}\, \mathcal{R}_1}{E \vdash \lambda \mathsf{x}.\, t : (\mathsf{x} : \mathcal{R}_1) \to^{\operatorname{dom} E} \mathcal{R}_2}$$

for a suitable notion of wellformedness for the relation of the argument. We defer the definition of wellformedness to §3. The issue is to define the arrow combinator from $\mathcal{R}_1$ to $\mathcal{R}_2$. To do this, we draw inspiration from logical relations: a denotation for a function must take any denotation for its arguments, and produce a denotation for its results. Since our relations have values on their right-hand sides, a sensible idea is to create the application of that value to some argument v_1, and take its normal form v_2. Following the same reasoning as for let-bindings, this means v_2 should be in the relation $\exists^{\operatorname{dom} E}[\mathsf{x}].\mathcal{R}_1\,[\mathsf{x} \mapsto v_1]\,;\mathcal{R}_2$. Contrarily to what we did for let-bindings, we have not closed the definition over v_1 by taking a union, to avoid losing precision. So far, our candidate definition for the combination of $\mathcal{R}_1$ and $\mathcal{R}_2$ for functions looks like this:

$$\left\{(\sigma, v) \;\middle|\; \begin{array}{l} \forall v_1, v_2, \sigma,\ v\, v_1 \rightsquigarrow^\star v_2 \Rightarrow \\ \quad (\sigma, v_1) \in \mathcal{R}_1 \Rightarrow (\sigma, v_2) \in \exists^{\operatorname{dom} E}[\mathsf{x}].\mathcal{R}_1\,[\mathsf{x} \mapsto v_1]\,;\mathcal{R}_2 \end{array}\right\}$$

This relation is, however, not stable. Our semantic framework is based on stable relations, and we have seen in §2.6 that stability is essential for soundness. A simple way to ensure stability is to strengthen the definition, by giving it a Kripke flavour. We simply take the intersection of such relations that are stable under extensions of substitutions. This leads to the following definition:

$$\left\{(\sigma, v) \;\middle|\; \begin{array}{l} \forall v_1, v_2,\ v\, v_1 \rightsquigarrow^\star v_2 \Rightarrow \forall \sigma',\ \sigma \sqsubseteq \sigma' \Rightarrow \\ \quad (\sigma', v_1) \in \mathcal{R}_1 \Rightarrow (\sigma', v_2) \in \exists^{\operatorname{dom} E}[\mathsf{x}].\mathcal{R}_1\,[\mathsf{x} \mapsto v_1]\,;\mathcal{R}_2 \end{array}\right\}$$

This is indeed a definition *à la* Kripke, since we can view our relations as sets of values that are indexed by substitutions. The usual quantification over larger worlds is, in our case, a quantification over all the extensions of substitutions. This quantification ensures the stability of the relation. For technical reasons—explained in §5.1—we further restrict the substitutions to those whose domains

are larger than $\mathrm{dom}\, E$. This leads to the combinator $\mathsf{FUN}^{\mathcal{S}}(\mathcal{R}, \mathcal{F})$, that takes a set of variables $\mathcal{S}$, a relation $\mathcal{R}$, and a function $\mathcal{F}$ from values to relations:

$$\mathsf{FUN}^{\mathcal{S}}(\mathcal{R}, \mathcal{F}) \triangleq \left\{ (\sigma, v) \;\middle|\; \begin{array}{l} \sigma \in \Sigma_v \wedge v \in \mathcal{V} \wedge \mathcal{S} \subseteq \mathrm{dom}\, \sigma \wedge \\ \forall v_1, v_2, \sigma',\ v\, v_1 \rightsquigarrow^\star v_2 \Rightarrow \sigma \sqsubseteq \sigma' \Rightarrow (\sigma', v_1) \in \mathcal{R} \Rightarrow (\sigma', v_2) \in \mathcal{F}(v_1) \end{array} \right\}$$

Finally, the interpretation of functions, that we call the *arrow* combinator, is defined as

$$(\mathsf{x} : \mathcal{R}_1) \to^{\mathcal{S}} \mathcal{R}_2 \triangleq \mathsf{FUN}^{\mathcal{S}}\Big(\mathcal{R}_1, \lambda v.\, \exists^{\mathcal{S}}[\mathsf{x}].\mathcal{R}_1\,[\mathsf{x} \mapsto v]\,; \mathcal{R}_2\Big)$$

The interpretation of applications is, in comparison, straightforward:

$$\frac{E \vdash t_1 : \mathcal{R}_1 \qquad E \vdash t_2 : \mathcal{R}_2}{E \vdash t_1\, t_2 : \mathsf{APP}(\mathcal{R}_1, \mathcal{R}_2)}$$

The rule interprets the two operands, and applies the first one to the other one, by considering the normal forms of the applications of the corresponding values: $\mathsf{APP}(\mathcal{R}_1, \mathcal{R}_2) \triangleq \{(\sigma, v) \mid \exists v_1, v_2, v_1\, v_2 \rightsquigarrow^\star v \wedge (\sigma, v_1) \in \mathcal{R}_1 \wedge (\sigma, v_2) \in \mathcal{R}_2\}$. We can show a property analogous to β-reduction at the level of relations: $\mathsf{APP}\big((\mathsf{x} : \mathcal{R}_1') \to^{\mathcal{S}} \mathcal{R}_2, \mathcal{R}_1\big) \subseteq \mathsf{LET}^{\mathcal{S}}\ \mathsf{x} \leftarrow \mathcal{R}_1\ \mathsf{IN}\ \mathcal{R}_2$, as soon as $\mathcal{R}_1 \subseteq \mathcal{R}_1'$.

2.8 Examples

For the identity function $\lambda \mathsf{x}.\, \mathsf{x}$, we can derive the relation $(\mathsf{x} : \top) \to^{\emptyset} \mathsf{SELF}(\mathsf{x})$, that effectively captures the extensional behaviour of the identity. Thanks to the notations we introduced, this closely resembles the definition of the identity function. Similarly, the boolean $\lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathsf{x}$ can be given the denotation $(\mathsf{x} : \top) \to^{\emptyset} (\mathsf{y} : \top) \to^{\{\mathsf{x}\}} \mathsf{SELF}(\mathsf{x})$. Unsurprisingly, the application functional $\lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\, \mathsf{x}$ is interpreted as $(\mathsf{f} : \top) \to^{\emptyset} (\mathsf{x} : \top) \to^{\{\mathsf{f}\}} \mathsf{APP}(\mathsf{SELF}(\mathsf{f}), \mathsf{SELF}(\mathsf{x}))$, which again looks like a rewriting of the definition.

Let us define $\Delta \triangleq \lambda \mathsf{x}.\, \mathsf{x}\, \mathsf{x}$, and $\Omega \triangleq \Delta\, \Delta$. The term Ω reduces to itself, and thus has no normal form. For Δ, we can derive the relation $\mathsf{SELF}(\Delta) = \{(\sigma, v) \mid \sigma \in \Sigma_v \wedge v = \Delta\}$, and we can derive the relation $\mathsf{APP}(\mathsf{SELF}(\Delta), \mathsf{SELF}(\Delta))$ for the term Ω. By unfolding the definitions, any $(\sigma, v) \in \mathsf{APP}(\mathsf{SELF}(\Delta), \mathsf{SELF}(\Delta))$ must also satisfy $\Delta\, \Delta \rightsquigarrow^\star v$. This is impossible, because Ω is known to diverge. Thus, there cannot be any element in $\mathsf{APP}(\mathsf{SELF}(\Delta), \mathsf{SELF}(\Delta))$. In other words, by applying the interpretation rules, we obtain for the diverging term Ω the empty relation—an expected result.

This example shows that ill-typed terms, that involve self applications, are also supported. Untyped recursion combinators based on self application, such as the Y combinator, are supported too, and thus we do not present any construct for explicit recursion at this moment. We defer to §4.4 the discussion on how to add native fixpoint constructs.

3 NAME TRANSPOSITIONS, SUPPORTING SETS, AND WELLFORMED RELATIONS

Now that we have walked the reader through the intuitions behind the interpretation of λ-terms, we give a more formal account for the nominal techniques [Gabbay and Pitts 1999; Pitts 2016] we have employed so far. Remember that we identified a technical subtlety in how our operators on relations should handle bound names. For example, we need to ensure that the choice of the variable x in the relation $\mathsf{LET}^{\mathcal{S}}\ \mathsf{x} \leftarrow \top\ \mathsf{IN}\ \mathsf{SELF}(\mathsf{x})$ does not matter, *i.e.* does not belong to the "free variables" of that relation. We use nominal techniques to define properly this notion of free variables for relations.

A central concept in nominal techniques is the one of name *permutations*, or *transpositions*. They are operations that exchange names for other ones, in a bijective manner. The transposition of x for y in the variable z is written $[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot \mathsf{z}$ and is defined as follows: $[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot \mathsf{x} = \mathsf{y}$, and $[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot \mathsf{y} = \mathsf{x}$, and $[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot \mathsf{z} = \mathsf{z}$ when $\mathsf{z} \neq \mathsf{x}$ and $\mathsf{z} \neq \mathsf{y}$. We can lift the transposition to terms, in a homomorphic way. For instance, $[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot (t_1\, t_2) = ([\mathsf{x} \leftrightarrow \mathsf{y}] \cdot t_1)\, ([\mathsf{x} \leftrightarrow \mathsf{y}] \cdot t_2)$, but also

Operation	Notation	Definition
Union	$\mathcal{R}_1 \cup \mathcal{R}_2$	$\{(x,y) \mid (x,y) \in \mathcal{R}_1 \vee (x,y) \in \mathcal{R}_2\}$
Intersection	$\mathcal{R}_1 \cap \mathcal{R}_2$	$\{(x,y) \mid (x,y) \in \mathcal{R}_1 \wedge (x,y) \in \mathcal{R}_2\}$
Composition	$\mathcal{R}_1 ; \mathcal{R}_2$	$\{(x,z) \mid \exists y, (x,y) \in \mathcal{R}_1 \wedge (y,z) \in \mathcal{R}_2\}$
Top	$\top$	$\Sigma_v \times \mathcal{V}$
Top for values	$\top_{\mathsf{v}}$	$\mathcal{V} \times \mathcal{V}$
Bottom	$\bot$	$\emptyset$
Equality	EQ	$\{(v,v) \mid v \in \mathcal{V}\}$
Selfification	$\mathsf{SELF}(t)$	$\{(\sigma, t \cdot \sigma) \mid \sigma \in \Sigma_v \wedge t \cdot \sigma \in \mathcal{V} \wedge \mathrm{fv}\, t \subseteq \mathrm{dom}\, \sigma\}$
Gathering	$\mathsf{GATHER}(E)$	$\left\{(\sigma, v) \middle\vert \begin{array}{l} \sigma \in \Sigma_v \wedge v \in \mathcal{V} \wedge \\ \forall x \in \mathrm{dom}\, E, x \in \mathrm{dom}\, \sigma \wedge (\sigma, \sigma(x)) \in E(x) \end{array}\right\}$
Push	$\mathcal{R}[\mathsf{x} \mapsto v]$	$\{(\sigma, \sigma[\mathsf{x} \mapsto v]) \mid (\sigma, v) \in \mathcal{R}\}$
Restriction	$\exists^{\mathcal{S}}[\mathsf{x}].\mathcal{R}$	$\bigcup_{y \notin \mathcal{S}} [y \leftrightarrow \mathsf{x}] \cdot \mathcal{R}$
Chaining	$\mathsf{LET}^{\mathcal{S}}\, \mathsf{x} \leftarrow \mathcal{R}_1\, \mathsf{IN}\, \mathcal{R}_2$	$\exists^{\mathcal{S}}[\mathsf{x}]. \bigcup_v (\mathcal{R}_1[\mathsf{x} \mapsto v] ; \mathcal{R}_2)$
Function	$\mathsf{FUN}^{\mathcal{S}}(\mathcal{R}, \mathcal{F})$	$\left\{(\sigma, f) \middle\vert \begin{array}{ll} \sigma \in \Sigma_v \wedge f \in \mathcal{V} \wedge \mathcal{S} \subseteq \mathrm{dom}\, \sigma \wedge & \\ \forall \sigma', v_1, v_2, & \sigma \sqsubseteq \sigma' \Rightarrow f\, v_1 \rightsquigarrow^{\star} v_2 \Rightarrow \\ & (\sigma', v_1) \in \mathcal{R} \Rightarrow (\sigma', v_2) \in \mathcal{F}(v_1) \end{array}\right\}$
Arrow	$(\mathsf{x} : \mathcal{R}_1) \rightarrow^{\mathcal{S}} \mathcal{R}_2$	$\mathsf{FUN}^{\mathcal{S}}\big(\mathcal{R}_1, \lambda v. \exists^{\mathcal{S}}[\mathsf{x}]. (\mathcal{R}_1[\mathsf{x} \mapsto v] ; \mathcal{R}_2)\big)$
Application	$\mathsf{APP}(\mathcal{R}_1, \mathcal{R}_2)$	$\big\{(\sigma, v) \mid \exists v_1, v_2, v_1\, v_2 \rightsquigarrow^{\star} v \wedge (\sigma, v_1) \in \mathcal{R}_1 \wedge (\sigma, v_2) \in \mathcal{R}_2\big\}$
Right-unit	$\mathsf{UNIT}^{\mathsf{R}}$	$\{(\sigma, ()) \mid \sigma \in \Sigma_v\}$
Left-pairing	$\mathsf{PAIR}^{\mathsf{L}}(\mathcal{R}_1, \mathcal{R}_2)$	$\{((v_1, v_2), v) \mid (v_1, v) \in \mathcal{R}_1 \wedge (v_2, v) \in \mathcal{R}_2\}$
Right-pairing	$\mathsf{PAIR}^{\mathsf{R}}(\mathcal{R}_1, \mathcal{R}_2)$	$\{(\sigma, (v_1, v_2)) \mid (\sigma, v_1) \in \mathcal{R}_1 \wedge (\sigma, v_2) \in \mathcal{R}_2\}$
Left-suming	$\mathsf{SUM}^{\mathsf{L}}(\mathcal{R}_1, \mathcal{R}_2)$	$\big\{(\mathsf{inj}_1\, v_1, v) \mid (v_1, v) \in \mathcal{R}_1\big\} \cup \big\{(\mathsf{inj}_2\, v_2, v) \mid (v_2, v) \in \mathcal{R}_2\big\}$
Right-suming	$\mathsf{SUM}^{\mathsf{R}}(\mathcal{R}_1, \mathcal{R}_2)$	$\big\{(\sigma, \mathsf{inj}_1\, v_1) \mid (\sigma, v_1) \in \mathcal{R}_1\big\} \cup \big\{(\sigma, \mathsf{inj}_2\, v_2) \mid (\sigma, v_2) \in \mathcal{R}_2\big\}$
Matching	$\mathsf{MATCH}^{\mathcal{S}}\ \mathcal{R}\ \mathsf{WITH}$ $\mid \mathsf{x}_1 \leftarrow \mathcal{R}_1 \mid \mathsf{x}_2 \leftarrow \mathcal{R}_2$	$(\mathsf{LET}^{\mathcal{S}}\ \mathsf{x}_1 \leftarrow \mathcal{R}; \mathsf{PAIR}^{\mathsf{L}}(\mathsf{EQ}, \bot)\ \mathsf{IN}\ \mathcal{R}_1) \cup$ $(\mathsf{LET}^{\mathcal{S}}\ \mathsf{x}_2 \leftarrow \mathcal{R}; \mathsf{PAIR}^{\mathsf{L}}(\bot, \mathsf{EQ})\ \mathsf{IN}\ \mathcal{R}_2)$

Fig. 2. Basic relations, and combinators of relations

$[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot (\lambda \mathsf{z}.\, t) = \lambda [\mathsf{x} \leftrightarrow \mathsf{y}] \cdot \mathsf{z}.\, [\mathsf{x} \leftrightarrow \mathsf{y}] \cdot t$. This means that bound variables—such as the z in $\lambda \mathsf{z}.\, t$—must be exchanged too. Transpositions enjoy several algebraic properties:

Involution: $[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot ([\mathsf{x} \leftrightarrow \mathsf{y}] \cdot t) = t$
Reflexivity: $[\mathsf{x} \leftrightarrow \mathsf{x}] \cdot t = t$
Symmetry: $[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot t = [\mathsf{y} \leftrightarrow \mathsf{x}] \cdot t$
Transitivity: $[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot ([\mathsf{y} \leftrightarrow \mathsf{z}] \cdot t) = [\mathsf{x} \leftrightarrow \mathsf{z}] \cdot t$ when $\mathsf{x} \notin \mathrm{fv}\, t$ and $\mathsf{y} \notin \mathrm{fv}\, t$
Composition: $[\mathsf{x}_1 \leftrightarrow \mathsf{x}_2] \cdot ([\mathsf{y}_1 \leftrightarrow \mathsf{y}_2] \cdot t) = [([\mathsf{x}_1 \leftrightarrow \mathsf{x}_2] \cdot \mathsf{y}_1) \leftrightarrow ([\mathsf{x}_1 \leftrightarrow \mathsf{x}_2] \cdot \mathsf{y}_2)] \cdot ([\mathsf{x}_1 \leftrightarrow \mathsf{x}_2] \cdot t)$

We can also lift transpositions to other syntactic objects, such as substitutions, in a homomorphic way. It is also possible to lift transpositions to *semantic* objects, such as relations, as follows: $[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot \mathcal{R} = \{(\sigma, v) \mid ([\mathsf{x} \leftrightarrow \mathsf{y}] \cdot \sigma, [\mathsf{x} \leftrightarrow \mathsf{y}] \cdot v) \in \mathcal{R}\}$. Thanks to the involution property, we have $(\sigma, v) \in \mathcal{R}$ iff $([\mathsf{x} \leftrightarrow \mathsf{y}] \cdot \sigma, [\mathsf{x} \leftrightarrow \mathsf{y}] \cdot v) \in [\mathsf{x} \leftrightarrow \mathsf{y}] \cdot \mathcal{R}$.

Free variables are also a significant concept, when reasoning about names. While free variables are easily defined on syntactic objects, devising a notion of free variables for semantic objects is challenging. Nominal techniques give an elegant solution: exchanging a non-free variable with another non-free variable should be the identity. This is precisely how we define *supporting sets*, following [Urban and Kaliszyk 2012].

Definition 3.1 (Supporting set). Assume that transposition is defined for an object O. We say that O is supported by a set of names $\mathcal{S}$, when $\mathcal{S}$ is finite and for every $\mathsf{x} \notin \mathcal{S}$ and $\mathsf{y} \notin \mathcal{S}$, $[\mathsf{x} \leftrightarrow \mathsf{y}] \cdot O = O$.

Supporting sets for terms are over-approximations of their free variables. More generally, a supporting set is a finite over-approximation of the *support*, which is the precise way of defining free variables in nominal techniques.

Lemma 3.2. *The term t is supported by $\mathcal{S}$ iff* $\mathrm{fv}\, t \subseteq \mathcal{S}$.

We can finally define what we mean by *wellformed relations*.

Definition 3.3 (Wellformed relation). A relation $\mathcal{R}$ is wellformed with respect to a finite set of variables $\mathcal{S}$, written $\mathrm{wf}_{\mathcal{S}}\, \mathcal{R}$, when $\mathcal{R} \in \wp(\Sigma_v \times \mathcal{V})$, and $\mathcal{R}$ is stable, and $\mathcal{R}$ is supported by $\mathcal{S}$.

Since the relation $\mathsf{SELF}(t)$ is supported by the free variables of t, we have $\mathrm{wf}_{\mathrm{fv}\, t}\, \mathsf{SELF}(t)$. Moreover, the combinators of relations of Fig. 2 preserve wellformedness. For example, composition preserves wellformedness: if $\mathrm{wf}_{\mathcal{S}}\, \mathcal{R}_1$ and $\mathrm{wf}_{\mathcal{S}}\, \mathcal{R}_2$, then $\mathrm{wf}_{\mathcal{S}}(\mathcal{R}_1; \mathcal{R}_2)$. As another example that involves binders, let-binding of relations also preserves wellformedness: if $\mathrm{wf}_{\mathcal{S}}\, \mathcal{R}_1$ and $\mathrm{wf}_{\{\mathsf{x}\} \cup \mathcal{S}}\, \mathcal{R}_2$ and $\mathsf{x} \notin \mathcal{S}$, then $\mathrm{wf}_{\mathcal{S}}(\mathsf{LET}^{\mathcal{S}}\ \mathsf{x} \leftarrow \mathcal{R}_1\ \mathsf{IN}\ \mathcal{R}_2)$. Consequently, x is a *bound* variable in the relation $\mathsf{LET}^{\mathcal{S}}\ \mathsf{x} \leftarrow \mathcal{R}_1\ \mathsf{IN}\ \mathcal{R}_2$.

4 PROGRAM LOGICS AND DENOTATIONAL SEMANTICS

We now give the complete set of rules that form our program logic, and we define the *denotation* of a program, that computes the least relation that is provable in the logic. We prove the denotation is sound and complete with respect to the operational semantics. Finally, we sketch how to extend the theory with recursive functions.

4.1 A Program Logic of Input-Output Relations

We define the rules of a program logic in Fig. 3. Environments are inductively defined by the grammar $E ::= \bullet \mid E, \mathsf{x} : \mathcal{R}$. The judgement $E \vdash t : \mathcal{R}$ says that under the assumptions on free variables represented by E, the input-output behaviour of the term t is over-approximated by $\mathcal{R}$. The set of inference rules is composed of two parts. The first part is syntax directed, while the other is not. The definition comprises the rules from §2, and a few new ones.

The rule Sub is a subsumption rule. It asserts that any relation can be replaced by a weaker one, as long as it is wellformed in the current environment. We also added the rule Inter, that takes the intersection of two relations. We also *extended* rule Self: while we had only introduced $\mathsf{SELF}(\mathsf{x})$, *i.e.* specialised to the case of variables, the use of $\mathsf{SELF}(t)$ remains valid when t is an *extended value*. The set of extended values $\mathcal{V}^+$ extends values with variables. It is inductively defined as follows:

$$w \quad ::= \quad \mathsf{x} \quad \mid \quad \lambda \mathsf{x}.\, t \quad \mid \quad () \quad \mid \quad (w, w) \quad \mid \quad \mathsf{inj}_1\, w \quad \mid \quad \mathsf{inj}_2\, w$$

As a sanity check, the rules of our program logic ensures that the induced relations are wellformed, provided that the environment contains only wellformed relations.

Definition 4.1. An environment E is wellformed, written $\mathrm{wf}\, E$, if $E = E_1, x : \mathcal{R}, E_2$ implies $\mathrm{wf}_{\mathrm{dom}\, E}\, \mathcal{R}$, and if E has no duplicate binding.

Lemma 4.2. *If* $\mathrm{wf}\, E$ *and* $E \vdash t : \mathcal{R}$, *then* $\mathrm{wf}_{\mathrm{dom}\, E}\, \mathcal{R}$.

$$\textsc{Var}\ \frac{\mathsf{x} \in \operatorname{dom} E}{E \vdash \mathsf{x} : E(\mathsf{x})} \qquad \textsc{Let}\ \frac{E \vdash t_1 : \mathcal{R}_1 \qquad E, \mathsf{x} : \mathcal{R}_1 \vdash t_2 : \mathcal{R}_2 \qquad \mathsf{x} \notin \operatorname{dom} E}{E \vdash \mathsf{let}\ \mathsf{x} = t_1\ \mathsf{in}\ t_2 : \mathsf{LET}^{\operatorname{dom} E}\ \mathsf{x} \leftarrow \mathcal{R}_1\ \mathsf{IN}\ \mathcal{R}_2}$$

$$\textsc{Lam}\ \frac{\mathrm{wf}_{\operatorname{dom} E}\, \mathcal{R}_1 \qquad \mathsf{x} \notin \operatorname{dom} E \qquad E, \mathsf{x} : \mathcal{R}_1 \vdash t_2 : \mathcal{R}_2}{E \vdash \lambda \mathsf{x}.\, t : (\mathsf{x} : \mathcal{R}_1) \to^{\operatorname{dom} E} \mathcal{R}_2}$$

$$\textsc{App}\ \frac{E \vdash t_1 : \mathcal{R}_1 \qquad E \vdash t_2 : \mathcal{R}_2}{E \vdash t_1\, t_2 : \mathsf{APP}(\mathcal{R}_1, \mathcal{R}_2)} \qquad \textsc{Unit}\ \frac{}{E \vdash () : \mathsf{UNIT}^{\mathsf{R}}} \qquad \textsc{Pair}\ \frac{E \vdash t_1 : \mathcal{R}_1 \qquad E \vdash t_2 : \mathcal{R}_2}{E \vdash (t_1, t_2) : \mathsf{PAIR}^{\mathsf{R}}(\mathcal{R}_1, \mathcal{R}_2)}$$

$$\textsc{Fst}\ \frac{E \vdash t : \mathcal{R}}{E \vdash \pi_1\, t : \mathcal{R}; \mathsf{PAIR}^{\mathsf{L}}(\mathsf{EQ}, \top_{\mathsf{v}})} \qquad \textsc{Snd}\ \frac{E \vdash t : \mathcal{R}}{E \vdash \pi_2\, t : \mathcal{R}; \mathsf{PAIR}^{\mathsf{L}}(\top_{\mathsf{v}}, \mathsf{EQ})} \qquad \textsc{InjL}\ \frac{E \vdash t : \mathcal{R}}{E \vdash \mathsf{inj}_1\, t : \mathsf{SUM}^{\mathsf{R}}(\mathcal{R}, \bot)}$$

$$\textsc{InjR}\ \frac{E \vdash t : \mathcal{R}}{E \vdash \mathsf{inj}_2\, t : \mathsf{SUM}^{\mathsf{R}}(\bot, \mathcal{R})}$$

$$\textsc{Match}\ \frac{\begin{array}{c} E \vdash t : \mathcal{R} \\ E, \mathsf{x}_1 : \mathcal{R}; \mathsf{PAIR}^{\mathsf{L}}(\mathsf{EQ}, \bot) \vdash t_1 : \mathcal{R}_1 \qquad \mathsf{x}_1 \notin \operatorname{dom} E \\ E, \mathsf{x}_2 : \mathcal{R}; \mathsf{PAIR}^{\mathsf{L}}(\bot, \mathsf{EQ}) \vdash t_2 : \mathcal{R}_2 \qquad \mathsf{x}_2 \notin \operatorname{dom} E \end{array}}{E \vdash \begin{array}{l} \mathsf{match}\ t\ \mathsf{with} \\ \mid \mathsf{inj}_1\, \mathsf{x}_1 \to t_1 \mid \mathsf{inj}_2\, \mathsf{x}_2 \to t_2 \end{array} : \begin{array}{l} \mathsf{MATCH}^{\operatorname{dom} E}\ \mathcal{R}\ \mathsf{WITH} \\ \mid \mathsf{x}_1 \leftarrow \mathcal{R}_1 \mid \mathsf{x}_2 \leftarrow \mathcal{R}_2 \end{array}}$$

$$\textsc{Sub}\ \frac{\mathrm{wf}_{\operatorname{dom} E}\, \mathcal{R}_2 \qquad E \vdash t : \mathcal{R}_1 \qquad \mathcal{R}_1 \subseteq \mathcal{R}_2}{E \vdash t : \mathcal{R}_2} \qquad \textsc{Inter}\ \frac{E \vdash t : \mathcal{R}_1 \qquad E \vdash t : \mathcal{R}_2}{E \vdash t : \mathcal{R}_1 \cap \mathcal{R}_2}$$

$$\textsc{Gather}\ \frac{E \vdash t : \mathcal{R}}{E \vdash t : \mathsf{GATHER}(E)} \qquad \textsc{Self}\ \frac{t \in \mathcal{V}^+ \qquad E \vdash t : \mathcal{R}}{E \vdash t : \mathsf{SELF}(t)}$$

Fig. 3. Inference rules of the program logic

Thus, if $E \vdash t : \mathcal{R}$, then the "free variables" of the relation $\mathcal{R}$ are necessarily in $\operatorname{dom} E$.

Our program logic for *untyped* λ-terms resembles a type system with subtyping and intersection types. In §4.2, we give an algorithm that produces the most precise relation, given a term and an environment. This most precise relation always exists. From a different viewpoint, this algorithm can also be understood as a denotational semantics of terms. We will show in §4.3 that this semantics is correct and complete with respect to the operational semantics. The use of an algorithm that computes a minimal solution is a standard proof technique to obtain completeness results. For example, the completeness proof of Floyd-Hoare logic [Cook 1978; Floyd 1993; Hoare 1969] exploits the construction of a weakest precondition. The equivalence of the two presentations—denotational *vs.* axiomatic—is also reminiscent of *domain theory in a logical form* [Abramsky 1991].

In the terminology of abstract interpretation, this semantics will be our *collecting semantics*, from which we will build abstractions and derive abstract interpreters. While the soundness of the collecting semantics is a necessary intermediate result to prove the soundness of the analyser, the completeness theorem ensures that no information is lost by abstracting from the collecting semantics rather than from the operational semantics.

4.2 Denoting Terms as Input-Output Relations

The denotation of a term t in an environment E, written $[\![t]\!]_E$, is a relation in $\wp(\Sigma_v \times \mathcal{V})$. It is defined in Fig. 4 by recursion on the syntax of terms, in the style of a denotational interpreter.

$$
\begin{aligned}
[\![\mathsf{x}]\!]_E &= \mathsf{SELF}(\mathsf{x}) \cap \mathsf{GATHER}(E) \quad \text{if } \mathsf{x} \in \operatorname{dom} E \\
[\![\mathsf{let}\ \mathsf{x} = t_1\ \mathsf{in}\ t_2]\!]_E &= \mathsf{LET}^{\operatorname{dom} E}\ \mathsf{x} \leftarrow [\![t_1]\!]_E\ \ \mathsf{IN}\ \ [\![t_2]\!]_{E,\mathsf{x}:[\![t_1]\!]_E} \text{ with } \mathsf{x} \notin \operatorname{dom} E \\
[\![\lambda \mathsf{x}.\, t]\!]_E &= ((\mathsf{x} : \top) \to^{\operatorname{dom} E} [\![t]\!]_{E,\mathsf{x}:\top}) \cap \mathsf{SELF}(\lambda \mathsf{x}.\, t) \cap \mathsf{GATHER}(E) \\
[\![t_1\ t_2]\!]_E &= \mathsf{APP}([\![t_1]\!]_E, [\![t_2]\!]_E) \\
[\![()]\!]_E &= \mathsf{UNIT}^{\mathsf{R}} \cap \mathsf{GATHER}(E) \\
[\![(t_1, t_2)]\!]_E &= \mathsf{PAIR}^{\mathsf{R}}([\![t_1]\!]_E, [\![t_2]\!]_E) \\
[\![\pi_1\ t]\!]_E &= [\![t]\!]_E ; \mathsf{PAIR}^{\mathsf{L}}(\mathsf{EQ}, \top_{\mathsf{v}}) \qquad [\![\pi_2\ t]\!]_E = [\![t]\!]_E ; \mathsf{PAIR}^{\mathsf{L}}(\top_{\mathsf{v}}, \mathsf{EQ}) \\
[\![\mathsf{inj}_1\ t]\!]_E &= \mathsf{SUM}^{\mathsf{R}}([\![t]\!]_E, \bot) \qquad [\![\mathsf{inj}_2\ t]\!]_E = \mathsf{SUM}^{\mathsf{R}}(\bot, [\![t]\!]_E) \\
\left[\!\!\left[\begin{array}{l}\mathsf{match}\ t\ \mathsf{with} \\ \mid\ \mathsf{inj}_1\ \mathsf{x}_1\ \to\ t_1 \\ \mid\ \mathsf{inj}_2\ \mathsf{x}_2\ \to\ t_2\end{array}\right]\!\!\right]_E &= \begin{array}{l}\mathsf{MATCH}^{\operatorname{dom} E}\ [\![t]\!]_E\ \ \mathsf{WITH} \\ \mid\ \mathsf{x}_1 \leftarrow [\![t_1]\!]_{E,\mathsf{x}_1:[\![t]\!]_E;\mathsf{SUM}^{\mathsf{L}}(\mathsf{EQ},\bot)} \text{ with } \mathsf{x}_1 \notin \operatorname{dom} E \text{ and } \mathsf{x}_2 \notin \operatorname{dom} E \\ \mid\ \mathsf{x}_2 \leftarrow [\![t_2]\!]_{E,\mathsf{x}_2:[\![t]\!]_E;\mathsf{SUM}^{\mathsf{L}}(\bot,\mathsf{EQ})}\end{array}
\end{aligned}
$$

Fig. 4. Denotation of a term

Most cases of the definition are the exact counterparts of the inference rules of Fig. 3. We review the other cases, that combine several inference rules by using intersection.

The interpretation of a variable x exploits Lemma 2.7 to improve on the expected result $E(\mathsf{x})$.

The interpretation of λ-abstractions combines the rules Lam, Self and Gather. The rule Lam uses $\top$ for the relation on the argument: in other words, it makes no assumption about the argument the function will be applied to. The use of the rules Gather and Self are necessary for completeness. One can understand the combination of Gather and Self as forming a *closure*: it specifies both the available contraints on the environment in which the function is defined, and the code of the function. The interpretation of functions thus mimics the usual evaluation rule found in a big-step semantics based on environments. Whereas the use of Lam gives an extensional flavour—it specifies the result of the function, given information on its argument—the rule Self brings some intensional information. It asserts, indeed, that we exactly know the *code* of the function. The presence of the two kinds of information—extensional and intensional—offers the possibility for static analysers to exploit one or the other, or both. The extensional part will give rise to modular analyses, that analyse the code of a function *once*, at its definition site, as found in type systems. The intensional part, however, allows a static analyser to track which functions are called, and in which environments, and permits an analyser to inline the code of functions when they are called, and improve precision. The static analysis of §5 only exploits the extensional, modular part.

The interpretation of the unit value combines the rules Unit and Gather, again for completeness reasons. We could have used rule Self instead of Unit, since $\mathsf{SELF}(()) = \mathsf{UNIT}^{\mathsf{R}}$.

Since the denotation of a term describes a strategy for applying the rules of the program logic, it is easy to show that the denotation of a term can be constructed using the rules of the logic. Moreover, the denotational semantics is always more precise than the program logic.

Lemma 4.3. *If* $\mathrm{wf}\ E$, *then* $E \vdash t : [\![t]\!]_E$. *Moreover,* $E \vdash t : \mathcal{R}$ *implies* $[\![t]\!]_E \subseteq \mathcal{R}$.

4.3 Soundness and Completeness

Our central result is that the denotational semantics *coincides* with the operational semantics. Its statement relies on the definition of the denotation of environments.

Definition 4.4 (Denotation of environments). Environments of relations denote sets of value substitutions. The denotation of an environment E, written $[\![E]\!]$, is inductively defined as follows:

$$
\frac{}{\bullet \in [\![\bullet]\!]} \qquad \frac{\sigma \in [\![E]\!] \qquad (\sigma, v) \in \mathcal{R} \qquad \mathsf{x} \notin \operatorname{dom} \sigma \vee \sigma(\mathsf{x}) = v}{\sigma\,[\mathsf{x} \mapsto v] \in [\![E, \mathsf{x} : \mathcal{R}]\!]} \qquad \frac{\sigma \in [\![E]\!] \qquad \sigma \sqsubseteq \sigma'}{\sigma' \in [\![E]\!]}
$$

The first rule of the definition asserts that the empty substitution $\bullet$ belongs to the denotation of the empty environment. The second rule follows the remark from §2.4: if σ is in the denotation of E and $(\sigma, v) \in \mathcal{R}$, then $\sigma\,[\mathsf{x} \mapsto v]$ is in the denotation of $E, \mathsf{x} : \mathcal{R}$. Additionally, we constrain that either $\mathsf{x} \notin \operatorname{dom}\sigma$ or that $\sigma(\mathsf{x}) = v$, so that the extended substitution $\sigma\,[\mathsf{x} \mapsto v]$ does not rebind former variables to different values. The third rule enforces that the set is closed under extensions of substitutions. The substitutions in $[\![E]\!]$ are at least defined on the domain of E itself.

Lemma 4.5. *If* $\sigma \in [\![E]\!]$, *then* $\operatorname{dom} E \subseteq \operatorname{dom}\sigma$.

Moreover, because the set of substitutions is upward closed, the following result holds:

Lemma 4.6. *If* $\sigma \in [\![E]\!]$ *and* $x \in \operatorname{dom} E$, *then* $(\sigma, \sigma(x)) \in E(x)$.

We are now ready to formally state our central theorem: for any wellformed environment of relations E that closes the term t, the denotation of t in E is equal to the input-output relation defined by t where its inputs are restricted to be in the denotation of E.

Theorem 4.7 (Adequacy). *Assume that* $\operatorname{fv} t \subseteq \operatorname{dom} E$ *and* $\operatorname{wf} E$. *Then,* $[\![t]\!]_E = (\!|t|\!)_{[\![E]\!]}$.

As a corollary, the program logic is both sound and complete with respect to the input-output relation of terms: the program logic always builds over-approximations of the input-output relation, and it is possible to derive exactly the input-output relation using the rules of the logic.

The proof of adequacy was mechanised in Coq. It is split into independent results for soundness and for completeness. Both are proved by induction over the definition of the denotation of terms. Each rule is proved independently, and relies on the monotonicity of the combinators and on local soundness and completeness results. For example, for let-bindings, the local soundness and completeness lemmas are the following.

Lemma 4.8 (Let soundness). *If* $\operatorname{fv} t_1 \subseteq \operatorname{dom} E$ *and* $\operatorname{fv} t_2 \subseteq \operatorname{dom}\{\mathsf{x}\} \cup E$ *and* $\mathsf{x} \notin \operatorname{dom} E$ *and* $\operatorname{wf} E$, *then* $(\!|\mathsf{let}\ \mathsf{x} = t_1\ \mathsf{in}\ t_2|\!)_{[\![E]\!]} \subseteq \mathsf{LET}^{\operatorname{dom} E}\ \mathsf{x} \leftarrow (\!|t_1|\!)_E\ \mathsf{IN}\ (\!|t_2|\!)_{E,\mathsf{x}:(\!|t_1|\!)_E}$.

Lemma 4.9 (Let completeness). *If* $\operatorname{fv} t_1 \subseteq \operatorname{dom} E$ *and* $\operatorname{fv} t_2 \subseteq \operatorname{dom}\{\mathsf{x}\} \cup E$ *and* $\mathsf{x} \notin \operatorname{dom} E$ *and* $\operatorname{wf} E$, *then* $\mathsf{LET}^{\operatorname{dom} E}\ \mathsf{x} \leftarrow (\!|t_1|\!)_E\ \mathsf{IN}\ (\!|t_2|\!)_{E,\mathsf{x}:(\!|t_1|\!)_E} \subseteq (\!|\mathsf{let}\ \mathsf{x} = t_1\ \mathsf{in}\ t_2|\!)_{[\![E]\!]}$.

The relations SELF (restricted to extended values) and GATHER are both sound, but only their intersection is complete. As we said already, the former specifies the right-hand side of the input-output relation only, whereas the latter specifies its left-hand side only.

Lemma 4.10 (Self soundness). *If* $\operatorname{fv} t \subseteq \operatorname{dom} E$ *and* $t \in \mathcal{V}^+$, *then* $(\!|t|\!)_{[\![E]\!]} \subseteq \mathsf{SELF}(t)$.

Lemma 4.11 (Gather soundness). *If* $\operatorname{fv} t \subseteq \operatorname{dom} E$ *and* $\operatorname{wf} E$, *then* $(\!|t|\!)_{[\![E]\!]} \subseteq \mathsf{GATHER}(E)$.

Lemma 4.12. *If* $\operatorname{fv} t \subseteq \operatorname{dom} E$ *and* $\operatorname{wf} E$, *then* $\mathsf{SELF}(t) \cap \mathsf{GATHER}(E) \subseteq (\!|t|\!)_{[\![E]\!]}$.

As a consequence, the combination of SELF and GATHER is both sound and complete for *extended values*. That combination is, however, not sound for arbitrary terms, that could be further reduced.

Another consequence is that $\mathsf{GATHER}(E) \cap \mathsf{SELF}(\lambda\mathsf{x}.\, t)$ is a sound and complete interpretation of $\lambda\mathsf{x}.\, t$ in the environment E, provided $\operatorname{fv}(\lambda\mathsf{x}.\, t) \subseteq \operatorname{dom} E$. The relation $(\mathsf{x} : \top) \to^{\operatorname{dom} E} [\![t]\!]_{E,\mathsf{x}:\top}$ is only guaranteed to be sound. This relation alone is, indeed, *not* complete. It has an extensional flavour, in the sense that if it has a term t' as element, then any η-expansion of t' also belongs to the relation. The relation $(\!|\lambda\mathsf{x}.\, t|\!)_{[\![E]\!]}$, however, is not closed under η-expansion. Thus, it cannot be the case that $(\mathsf{x} : \top) \to^{\operatorname{dom} E} [\![t]\!]_{E,\mathsf{x}:\top} \subseteq (\!|\lambda\mathsf{x}.\, t|\!)_{[\![E]\!]}$. The closure of $(\!|t|\!)$ under observational equivalence, and the study of the completeness of the arrow combinator are deferred to future work.

4.4 Handling Recursive Functions

We can already encode fixpoint combinators in our untyped language. Such combinators are not definable in typed languages, however. Instead, typed languages provide constructs to define recursive functions, so we might want to analyse these constructs too. We briefly present an extension with recursive functions. Although we formalised this extension and proved it correct on paper, we have not *mechanised* it, since doing so raises technical issues in a system like Coq. Therefore, we prefer to dedicate a specific section for the support of recursive functions.

Terms are augmented with a fixpoint construct $\mu\mathsf{f}.\,\lambda\mathsf{x}.\,t$. These fixpoints are values, and the operational semantics unfolds them on demand, when they get applied to a value.

$$t ::= \ldots \mid \mu\mathsf{f}.\,\lambda\mathsf{x}.\,t \qquad v ::= \ldots \mid \mu\mathsf{f}.\,\lambda\mathsf{x}.\,t \qquad (\mu\mathsf{f}.\,\lambda\mathsf{x}.\,t)\,v \rightsquigarrow (\lambda\mathsf{x}.\,t)\,[\mathsf{f} \leftarrow \mu\mathsf{f}.\,t]\;v$$

The denotation of fixpoints is given by $(\!|\mu\mathsf{f}.\,\lambda\mathsf{x}.\,t|\!)_E = \mathcal{R}_{\text{base}} \cap \operatorname{lfp} f$, where $\mathcal{R}_{\text{base}} = \mathsf{GATHER}(E) \cap \mathsf{SELF}(\mu\mathsf{f}.\,\lambda\mathsf{x}.\,t)$ and where $f(\mathcal{R}) = \mathcal{R}_{\text{base}} \cup \mathsf{FUN}^{\operatorname{dom} E}(\top, \lambda v.\mathsf{APP}(\mathcal{R}, \mathsf{SELF}(v))) \cup \mathsf{LET}^{\operatorname{dom} E}\ \mathsf{f} \leftarrow \mathcal{R}\ \mathsf{IN}\ (\!|\lambda\mathsf{x}.\,t|\!)_{E,\mathsf{f}:\mathcal{R}}$.

The combination of GATHER and SELF is already sound and complete, because $\mu\mathsf{f}.\,\lambda\mathsf{x}.\,t$ is a value. We can also show that any relation $\mathcal{R}$ is a sound approximation, as soon as $\mathrm{wf}_{\operatorname{dom} E}\,\mathcal{R}$, and $f(\mathcal{R}) \subseteq \mathcal{R}$. Thus, it suffices to find a post-fixpoint of the monotone function f. In particular, the least post-fixpoint of f, *i.e.* its least fixpoint, is a sound approximation. The relation $\mathsf{FUN}^{\operatorname{dom} E}(\top, \lambda v.\mathsf{APP}(\mathcal{R}, \mathsf{SELF}(v)))$ ensures that the relation $\mathcal{R}$ is closed under η-expansion of functions, therefore contains all the unfoldings of the fixpoint.

5 ABSTRACTING THE DENOTATION INTO A STATIC ANALYSIS

The goal of this section is to demonstrate the usefulness of our denotational semantics, by deriving from it a static analyser. To that end, we do the exercise of following the methodology of abstract interpretation, and systematically abstract the elements of the denotational semantics. We obtain an analysis that is *sound by construction*.

Our construction is articulated in two stages. First, we apply a variant of the *independent attribute* abstraction [Jones and Muchnick 1980] to transform relations in $\wp(\Sigma_v \times \mathcal{V})$ to mappings in $\mathsf{Var} \to (\mathcal{V} \times \mathcal{V})$. At this point, we know *how each input of a program is (independently) related to the output value*. The second stage of our construction abstracts relations in $\mathcal{V} \times \mathcal{V}$ into *correlations* [Andreescu et al. 2019]. The original domain of correlations abstracts relations over first-order values only. Guided by the first abstraction, we extend correlations to support functions as values.

5.1 A First Abstraction: Pointwise Binary Relations

To build our example analyser, we perform as a first stage a pointwise abstraction of substitutions. We write $A \xrightarrow{\text{fin}} B$ for finite maps from A to B and $A \xrightarrow{\text{fin}_0} B$ for the non-empty finite maps. We write $A + \mathbf{1}$ to denote the tagged sum of A with the one-element lattice, and write $A \times \mathbf{2}$ for the lattice product of A with the two-element lattice. We exploit the following calculation, where $\simeq$ stands for the presence of an isomorphism, and $\precsim$ for a Galois connection:

$$\begin{aligned}
\wp(\Sigma_v \times \mathcal{V}) &\simeq \mathcal{V} \to \wp(\mathsf{Var} \xrightarrow{\text{fin}} \mathcal{V}) \\
&\simeq \mathcal{V} \to \wp((\mathsf{Var} \xrightarrow{\text{fin}_0} \mathcal{V}) + \mathbf{1}) \\
&\simeq \mathcal{V} \to (\wp(\mathsf{Var} \xrightarrow{\text{fin}_0} \mathcal{V}) \times \mathbf{2}) \\
&\precsim \mathcal{V} \to ((\mathsf{Var} \xrightarrow{\text{fin}_0} \wp(\mathcal{V})) \times \mathbf{2}) \\
&\simeq (\mathcal{V} \to (\mathsf{Var} \xrightarrow{\text{fin}_0} \wp(\mathcal{V}))) \times (\mathcal{V} \to \mathbf{2}) \\
&\simeq (\mathsf{Var} \xrightarrow{\text{fin}_0} \wp(\mathcal{V} \times \mathcal{V})) \times \wp(\mathbf{1} \times \mathcal{V})
\end{aligned}$$

$$\frac{}{\bot^\natural \sqsubseteq^\natural \mathcal{M}} \qquad \frac{\mathcal{S}_1 = \mathcal{S}_2 \qquad \forall \mathsf{x} \in \mathcal{S}_1, \mathcal{R}^1_\mathsf{x} \subseteq \mathcal{R}^2_\mathsf{x} \qquad \mathcal{R}^1_{\mathsf{this}} \subseteq \mathcal{R}^2_{\mathsf{this}}}{\{\overline{\mathsf{x} \mapsto \mathcal{R}^1_\mathsf{x}}^{\mathsf{x} \in \mathcal{S}_1}, \mathsf{this} \mapsto \mathcal{R}^1_{\mathsf{this}}\} \sqsubseteq^\natural \{\overline{\mathsf{x} \mapsto \mathcal{R}^2_\mathsf{x}}^{\mathsf{x} \in \mathcal{S}_2}, \mathsf{this} \mapsto \mathcal{R}^2_{\mathsf{this}}\}}$$

Fig. 5. Inclusion on pointwise relations

$$\gamma_\mathcal{S}(\bot^\natural) = \bot$$
$$\gamma_\mathcal{S}(\{\overline{\mathsf{x} \mapsto \mathcal{R}_\mathsf{x}}^{\mathsf{x} \in \mathcal{S}'}, \mathsf{this} \mapsto \mathcal{R}_{\mathsf{this}}\}) = \left\{(\sigma, v) \;\middle|\; \begin{array}{l} \sigma \in \Sigma_v \wedge v \in \mathcal{V} \wedge \mathcal{S} \subseteq \operatorname{dom}\sigma \wedge \mathcal{S} = \mathcal{S}' \wedge \\ (\forall \mathsf{x} \in \mathcal{S}, (\sigma(\mathsf{x}), v) \in \mathcal{R}_\mathsf{x}) \wedge (\forall v' \in \mathcal{V}, (v', v) \in \mathcal{R}_{\mathsf{this}}) \end{array}\right\}$$
$$\alpha_\mathcal{S}(\mathcal{R}) = \text{if } \mathcal{R} = \bot \text{ then } \bot^\natural \text{ else } \{\overline{\mathsf{x} \mapsto \mathcal{R}_\mathsf{x}}^{\mathsf{x} \in \mathcal{S}}, \mathsf{this} \mapsto \mathcal{R}_{\mathsf{this}}\}$$
$$\text{where } \mathcal{R}_\mathsf{x} = \bigcup_{\sigma \in \Sigma_v} \left\{(\sigma(\mathsf{x}), v) \;\middle|\; \mathcal{S} \subseteq \operatorname{dom}\sigma \wedge (\sigma, v) \in \mathcal{R}\right\}$$
$$\text{and } \mathcal{R}_{\mathsf{this}} = \bigcup_{\sigma \in \Sigma_v} \left\{(v', v) \;\middle|\; \mathcal{S} \subseteq \operatorname{dom}\sigma \wedge v' \in \mathcal{V} \wedge (\sigma, v) \in \mathcal{R}\right\}$$

Fig. 6. Concretisation and abstraction between relations in $\wp(\Sigma_v \times \mathcal{V})$ and pointwise relations

The calculation first indexes the set of input substitutions over the output value. Then, we split the cases where the substitution is empty or not. The next step, with symbol $\precsim$, is where the independent attribute abstraction happens. Then, we distribute the indexing over output values on the two cases of empty and non-empty substitutions. Finally, we reorder the indexing, so that we start with indexing over input variables.

We obtain *pointwise binary relations*. They are composed of two pieces: a finite map from input variables to binary relations over values, and a set of (output) values. The first part is a map that tells how each input value (identified by the free variable it is bound to) is related to the output value. The second part keeps some information when the program we want to analyse has no free variables. We choose to represent the set of output values as a binary relation whose left-hand side carries no information, so that we uniformly handle binary relations all way through.

We denote pointwise binary relations by $\mathcal{M}$. They are described by the following syntax:

$$\mathcal{M} \quad ::= \quad \bot^\natural \quad | \quad \{\overline{\mathsf{x} \mapsto \mathcal{R}_\mathsf{x}}^{\mathsf{x} \in \mathcal{S}}, \mathsf{this} \mapsto \mathcal{R}_{\mathsf{this}}\}$$

where $\mathcal{R}_\mathsf{x}$ and $\mathcal{R}_{\mathsf{this}}$ are binary relations over values, for every x. For convenience, we represent our domain with only one mapping, using the distinguished variable this. The variable this is bound to the relation that denotes the set of output values (and whose left-hand side carries no information).

We use the notation $\mathcal{M}(\mathsf{x})$ to retrieve the relation found at index x in the mapping in $\mathcal{M}$. As a convention, it returns $\top_\mathsf{v}$ if there is no binding for x, and it returns $\bot$ if $\mathcal{M} = \bot^\natural$.

We define the inclusion of pointwise relations in Fig. 5 as the pointwise extension of inclusion of binary relations. Two comparable pointwise relation necessarily have maps with identical domains.

Lemma 5.1. *The relation $\sqsubseteq^\natural$ on pointwise relation is a preorder: it is a reflexive, transitive relation.*

Fig. 6 defines the concretisation and abstraction that result from the composition of the calculation steps we performed at the start of the section. They are parameterised by a set of variables $\mathcal{S}$, that denotes the minimal domains of the substitutions we are interested in. Following the principles of abstract interpretation [Cousot and Cousot 1977], the concretisation and abstraction functions form a Galois connection, and the abstract preorder is sound with respect to the concretisation.

Lemma 5.2 (Galois connection). *Let $\mathcal{R} \in \wp(\Sigma_v \times \mathcal{V})$ be a stable relation such that $\mathcal{S} \subseteq \operatorname{dom}\sigma$ as soon as $(\sigma, v) \in \mathcal{R}$. Then, $\alpha_\mathcal{S}(\mathcal{R}) \sqsubseteq^\natural \mathcal{M} \Leftrightarrow \mathcal{R} \subseteq \gamma_\mathcal{S}(\mathcal{M})$.*

The adjunction property only holds for relations that only contain substitutions whose domains are larger than $\mathcal{S}$. This is why, in §2.7, we enforced that the FUN combinator enjoyed this property.

Lemma 5.3 (Correctness of preorder). *If $\mathcal{M}_1 \sqsubseteq^\natural \mathcal{M}_2$, then $\gamma_\mathcal{S}(\mathcal{M}_1) \subseteq \gamma_\mathcal{S}(\mathcal{M}_2)$.*

Following the principles of abstract interpretation again, by simplifying the expression $\gamma_\mathcal{S} \circ f \circ \alpha_\mathcal{S}$, we get an abstract operator $f^\natural$ that over-approximates the concrete operator f. We obtain the abstract operations of Fig. 7, that are sound approximations of the operators on stable relations.

Lemma 5.4 (Correctness lemmas (excerpts)). *The operators of Fig. 7 are sound approximations of their corresponding concrete operators. For instance, the following assertions hold:*

- $\gamma_\mathcal{S}(\mathcal{M}_1) \cup \gamma_\mathcal{S}(\mathcal{M}_2) \subseteq \gamma_\mathcal{S}(\mathcal{M}_1 \sqcup^\natural \mathcal{M}_2)$
- $\gamma_\mathcal{S}(\mathcal{M}_1) \cap \gamma_\mathcal{S}(\mathcal{M}_2) \subseteq \gamma_\mathcal{S}(\mathcal{M}_1 \sqcap^\natural \mathcal{M}_2)$
- *If $\mathcal{R} \in \mathcal{V} \times \mathcal{V}$, then* $\gamma_\mathcal{S}(\mathcal{M}) ; \mathcal{R} \subseteq \gamma_\mathcal{S}(\mathcal{M} \fatsemi^\natural \mathcal{R})$
- $\mathsf{PAIR}^\mathsf{R}\left(\gamma_\mathcal{S}(\mathcal{M}_1), \gamma_\mathcal{S}(\mathcal{M}_2)\right) \subseteq \gamma_\mathcal{S}\left(\mathsf{PAIR}^{\natural\mathsf{R}}(\mathcal{M}_1, \mathcal{M}_2)\right)$
- $\mathsf{SUM}^\mathsf{R}\left(\gamma_\mathcal{S}(\mathcal{M}_1), \gamma_\mathcal{S}(\mathcal{M}_2)\right) \subseteq \gamma_\mathcal{S}\left(\mathsf{SUM}^{\natural\mathsf{R}}(\mathcal{M}_1, \mathcal{M}_2)\right)$
- *If $\mathsf{x} \notin \mathcal{S}$, then* $\mathsf{LET}^\mathcal{S}\ \mathsf{x} \leftarrow \gamma_\mathcal{S}(\mathcal{M}_1)\ \mathsf{IN}\ \gamma_{\{\mathsf{x}\}\cup\mathcal{S}}(\mathcal{M}_2) \subseteq \gamma_\mathcal{S}\left(\mathsf{LET}^\natural\ \mathsf{x} \leftarrow \mathcal{M}_1\ \mathsf{IN}\ \mathcal{M}_2\right)$

Most operations are abstracted pointwise. We use the standard functions map and map_2 on finite maps, that apply a function to all elements of a map—in the case of map_2, the operands must have the same domains. We also write $\mathcal{M}\setminus\mathsf{x}$ to denote the removal of the binding for x from the map.

The abstract operator $\mathsf{LET}^\natural\ \mathsf{x} \leftarrow \mathcal{M}_1\ \mathsf{IN}\ \mathcal{M}_2$ deserves some attention: when none of its operands are $\bot^\natural$, it is a map that associates every y to the relation $(\mathcal{M}_1(\mathsf{y}); \mathcal{M}_2(\mathsf{x})) \cap \mathcal{M}_2(\mathsf{y})$. Indeed we know by definition of the concretisation that $\mathcal{M}_1(\mathsf{y})$ relates $\sigma(\mathsf{y})$ to the intermediate value v computed by the let-binding. That very same value v is related to the output by $\mathcal{M}_2(\mathsf{x})$, since $\sigma[\mathsf{x} \mapsto v](\mathsf{x}) = v$. Thus, $\sigma(\mathsf{y})$ is related to the output value by the composition $\mathcal{M}_1(\mathsf{y}); \mathcal{M}_2(\mathsf{x})$. Moreover, we also know that $\sigma(\mathsf{y})$ is related to the output value by $\mathcal{M}_2(\mathsf{y})$. Hence, $\sigma(\mathsf{y})$ is related to the output value by the intersection of the two aforementioned relations.

The relation $\mathsf{SELF}(\mathsf{x})$ has an interesting abstraction: it is a mapping that maps x to EQ, and any other variable to $\top_\mathsf{v}$. Indeed, it represents the information that the output value *is* the value bound to the variable x. The independent attribute abstraction lost some information, though: only the binding for x remembers that the right-hand side is equal to x. The other bindings cannot express that information, because it would depend on some other binding, namely the binding for x.

Let us write $\mathcal{R}^{-1} \triangleq \{(x, y) \mid (y, x) \in \mathcal{R}\}$ to denote the converse of $\mathcal{R}$. Given an environment $E^\natural = \{\overline{\mathsf{x} \mapsto \mathcal{M}}\}$ of pointwise relations, the abstraction of GATHER computes, for every $\mathsf{x} \in \mathrm{dom}\, E^\natural$, the most precise information about the input x. This information is contained in the left-hand sides of $E^\natural(\mathsf{y})(\mathsf{x})$ and of $E^\natural(\mathsf{x})(\mathsf{y})^{-1}$, since they relate the input x with the input y, and in the left-hand side of $E^\natural(\mathsf{x})(\mathsf{this})^{-1}$, because is relates the input x to any other value. The definition of $\mathsf{GATHER}^\natural$ takes the intersection of the aforementioned relations.

The abstraction of the application combinator, written $\mathsf{APP}^\natural(\mathcal{M}_1, \mathcal{M}_2)$, returns $\bot^\natural$ as soon as one of the arguments is $\bot^\natural$, because we considered an eager semantics. We would have obtained a different definition, had we chosen a lazy evaluation semantics. When the arguments are not $\bot^\natural$, the abstraction is defined pointwise, using the $\mathsf{APP}_v(\mathcal{R}_1, \mathcal{R}_2)$ operator, defined in Fig. 8. It is similar to the application combinator we introduced in §2.7. $\mathsf{APP}_v(\mathcal{R}_1, \mathcal{R}_2)$ relates a value v with a value v' such that v' is the normal form of some application $v_1\, v_2$ where $(v, v_1) \in \mathcal{R}_1$ and $(v, v_2) \in \mathcal{R}_2$.

The abstraction of the FUN combinator raises a methodological issue. As expected, this combinator is *covariant* in its second argument, and *contravariant* in its first argument. Consequently,

to compute an over-approximation, we should over-approximate its second argument, but we also need to *under*-approximate its first argument. This is beyond our reach, since the abstraction/concretisation pair only deals with over-approximations. To devise a sound definition for $\mathsf{FUN}^{\natural S}(\mathsf{x}, \mathcal{M}_1, \mathcal{M}_2)$, we could therefore not apply the methodology recommended by the abstract interpretation theory. We found a different solution: we impose that $\mathcal{M}_1$ cannot constrain the input substitutions at all. For example, this constraint is always satisfied when $\mathcal{M}_1$ is the mapping that binds all variables to $\top_{\mathsf{v}}$. In more technical terms, the constraint imposes that the relation for the argument must convey no information on its left-hand side. This means that the argument of the closure cannot assume anything on the values of the environment in which the closure was created. While this still allows to impose some precondition on the argument—like being an integer greater than 42—this constraint disallows, however, *relational* preconditions—like being an integer greater than the value of some z present in the context where the closure is defined. This is no surprise: it is a consequence of the independent attribute abstraction. Dealing with the contravariance of arguments using *under*-approximations is left to future work.

Leaving apart the story about contravariance, the abstraction of the arrow constructor naturally introduces a *ternary* combinator on relations, in which the *relation between the argument of a function and its results* comes into play. The relation that is associated to every variable y is $\mathsf{FUN}^{\mathsf{R}}_{v}\left(h(\mathcal{M}_1(\mathsf{y}))(\mathcal{M}_1),\ \mathcal{M}_2(\mathsf{x}),\ (\mathcal{M}_1(\mathsf{y}); \mathcal{M}_2(\mathsf{x})) \cap \mathcal{M}_2(\mathsf{y})\right)$. The definition refers to the combinator $\mathsf{FUN}^{\mathsf{R}}_{v}\left(\mathcal{R}_{\text{arg}}, \mathcal{R}_{\text{inner}}, \mathcal{R}_{\text{outer}}\right)$ that is defined in Fig. 8. This combinator relates values v and v' such that for any input v_1 such that $(v, v_1) \in \mathcal{R}_{\text{arg}}$, the normal form v_2 of $v'\ v_1$ must satisfy $(v, v_2) \in \mathcal{R}_{\text{outer}}$ and $(v_1, v_2) \in \mathcal{R}_{\text{inner}}$. The relation $\mathcal{R}_{\text{arg}}$ relates an input to the argument, whereas the relation $\mathcal{R}_{\text{outer}}$ relates the same input to the result of the function applied to the input. The relation $\mathcal{R}_{\text{inner}}$ describes the input-output behaviour of the function. This extensional behaviour is precisely described by the relation $\mathcal{M}_2(\mathsf{x})$, since it relates the value that will be given to the variable x when the function is called, to the result of the call. Therefore we instantiate $\mathcal{R}_{\text{inner}}$ as $\mathcal{M}_2(\mathsf{x})$. The relation $\mathcal{R}_{\text{outer}}$ is instantiated with $(\mathcal{M}_1(\mathsf{y}); \mathcal{M}_2(\mathsf{x})) \cap \mathcal{M}_2(\mathsf{y})$, which is also the expression used in the abstraction of let-bindings. Finally, we choose $h(\mathcal{M}_1(\mathsf{y}))(\mathcal{M}_1)$ to instantiate $\mathcal{R}_{\text{arg}}$. In the simpler case where where $\mathcal{M}_1$ always binds variables to $\top_{\mathsf{v}}$, this is equivalent to choosing $\mathcal{R}_{\text{arg}} = \mathcal{M}_1(\mathsf{y})$.

5.2 A Second Abstraction: Correlations

In this section, we abstract binary relations on values into elements of the abstract domain of *correlations* [Andreescu et al. 2019]. We apply this abstraction in a functorial way to transform the pointwise relations of §5.1 into pointwise correlations (where binary relations are replaced with correlations).

The *original* domain of correlations is an abstraction of binary relations over first-order typed values. It can deal with product and sum values, but not with functions as values. Based on the ternary combinator for functions that we introduced in §5.1, we extend the domain of correlations with a case for functions. Finding this ternary combinator was the difficult part of the extension, and the guide provided by the previous independent attribute abstraction was of great help. Once we settled with this combinator, extending the correlation domain was not technically challenging.

The abstract domain of correlations is merely a *syntax* for the combinators on relations over values that we has seen so far:

$$C \quad ::= \quad \bot^\sharp \quad | \quad \mathsf{EQ}^\sharp \quad | \quad \top^\sharp \quad | \quad \mathsf{PAIR}^{\sharp S}(C, C) \quad | \quad \mathsf{SUM}^{\sharp S}(C, C) \quad | \quad \mathsf{FUN}^{\sharp S}(C, C, C)$$

Correlations C have three base cases: the bottom relation, the equality relation, and the top relation. We can also build compound correlations using a constructor for pairs, for sums, and for functions. The constructors hold a *side* S, which can be L (left) or R (right). The side indicates on which side of the binary relation the pair (resp. sum, function) should be.

$$
\begin{aligned}
\mathcal{M} \sqcup^\natural \bot^\natural &= \bot^\natural \sqcup^\natural \mathcal{M} = \mathcal{M} \\
\mathcal{M}_1 \sqcup^\natural \mathcal{M}_2 &= \mathrm{map}_2\,(\lambda(\mathcal{R}_1,\mathcal{R}_2).\,\mathcal{R}_1 \cup \mathcal{R}_2)\,\mathcal{M}_1\,\mathcal{M}_2 \text{ if } \mathcal{M}_1 \neq \bot^\natural \text{ and } \mathcal{M}_2 \neq \bot^\natural \\
\mathcal{M} \sqcap^\natural \bot^\natural &= \bot^\natural \sqcap^\natural \mathcal{M} = \bot^\natural \\
\mathcal{M}_1 \sqcap^\natural \mathcal{M}_2 &= \mathrm{map}_2\,(\lambda(\mathcal{R}_1,\mathcal{R}_2).\,\mathcal{R}_1 \cap \mathcal{R}_2)\,\mathcal{M}_1\,\mathcal{M}_2 \text{ if } \mathcal{M}_1 \neq \bot^\natural \text{ and } \mathcal{M}_2 \neq \bot^\natural \\
\bot^\natural \fatsemi^\natural \mathcal{R} &= \bot^\natural \\
\mathcal{M} \fatsemi^\natural \mathcal{R} &= \mathrm{map}\,(\lambda\mathcal{R}'.\,\mathcal{R}';\mathcal{R})\,\mathcal{M} \text{ if } \mathcal{M} \neq \bot^\natural \\
\mathsf{PAIR}^{\natural\mathsf{R}}\left(\mathcal{M}, \bot^\natural\right) &= \mathsf{PAIR}^{\natural\mathsf{R}}\left(\bot^\natural, \mathcal{M}\right) = \bot^\natural \\
\mathsf{PAIR}^{\natural\mathsf{R}}\left(\mathcal{M}_1, \mathcal{M}_2\right) &= \mathrm{map}_2\,(\lambda(\mathcal{R}_1,\mathcal{R}_2).\mathsf{PAIR}^{\mathsf{R}}\,(\mathcal{R}_1,\mathcal{R}_2))\,\mathcal{M}_1\,\mathcal{M}_2 \text{ if } \mathcal{M}_1 \neq \bot^\natural \text{ and } \mathcal{M}_2 \neq \bot^\natural \\
\mathsf{SUM}^{\natural\mathsf{R}}\left(\bot^\natural, \bot^\natural\right) &= \bot^\natural \\
\mathsf{SUM}^{\natural\mathsf{R}}\left(\mathcal{M}, \bot^\natural\right) &= \mathrm{map}\,(\lambda\mathcal{R}.\mathsf{SUM}^{\mathsf{R}}\,(\mathcal{R},\bot))\,\mathcal{M} \text{ if } \mathcal{M} \neq \bot^\natural \\
\mathsf{SUM}^{\natural\mathsf{R}}\left(\bot^\natural, \mathcal{M}\right) &= \mathrm{map}\,(\lambda\mathcal{R}.\mathsf{SUM}^{\mathsf{R}}\,(\bot,\mathcal{R}))\,\mathcal{M} \text{ if } \mathcal{M} \neq \bot^\natural \\
\mathsf{SUM}^{\natural\mathsf{R}}\left(\mathcal{M}_1, \mathcal{M}_2\right) &= \mathrm{map}_2\,(\lambda(\mathcal{R}_1,\mathcal{R}_2).\mathsf{SUM}^{\mathsf{R}}\,(\mathcal{R}_1,\mathcal{R}_2))\,\mathcal{M}_1\,\mathcal{M}_2 \text{ if } \mathcal{M}_1 \neq \bot^\natural \text{ and } \mathcal{M}_2 \neq \bot^\natural \\
\mathsf{LET}^\natural\ \mathsf{x} \leftarrow \mathcal{M}\ \mathsf{IN}\ \bot^\natural &= \mathsf{LET}^\natural\ \mathsf{x} \leftarrow \bot^\natural\ \mathsf{IN}\ \mathcal{M} = \bot^\natural \\
\mathsf{LET}^\natural\ \mathsf{x} \leftarrow \mathcal{M}_1\ \mathsf{IN}\ \mathcal{M}_2 &= \mathrm{map}_2\,(\lambda(\mathcal{R}_1,\mathcal{R}_2).\,(\mathcal{R}_1;\mathcal{M}_2(\mathsf{x})) \cap \mathcal{R}_2)\,\mathcal{M}_1\,(\mathcal{M}_2 \setminus \mathsf{x}) \\
&\quad \text{if } \mathcal{M}_1 \neq \bot^\natural \text{ and } \mathcal{M}_2 \neq \bot^\natural \text{ and } \mathsf{x} \in \mathrm{dom}\,\mathcal{M}_2 \\
\mathsf{SELF}^\natural_{\mathcal{S}}(\mathsf{x}) &= \{\overline{\mathsf{y} \mapsto \delta_{\mathsf{x}}(\mathsf{y})}^{\mathsf{y}\in\mathcal{S}}, \mathsf{this} \mapsto \top_{\mathsf{v}}\} \text{ where } \delta_{\mathsf{x}}(\mathsf{y}) = \text{if } \mathsf{y} = \mathsf{x} \text{ then } \mathsf{EQ} \text{ else } \top_{\mathsf{v}} \\
\mathsf{GATHER}^\natural\left(E^\natural\right) &= \{\overline{\mathsf{x} \mapsto G(\mathsf{x})}^{\mathsf{x}\in\mathrm{dom}\,E^\natural}, \mathsf{this} \mapsto \top_{\mathsf{v}}\} \\
\text{where } G(x) &= \left(\textstyle\bigcap_{y\in\mathrm{dom}\,E^\natural\setminus x}(E^\natural(y)(x);\top_{\mathsf{v}})\right) \cap \left(\textstyle\bigcap_{z\in\mathrm{dom}\,E^\natural(x)}(E^\natural(x)(z)^{-1};\top_{\mathsf{v}})\right) \cap E^\natural(x)(\mathsf{this})^{-1} \\
\mathsf{APP}^\natural\left(\mathcal{M}, \bot^\natural\right) &= \mathsf{APP}^\natural\left(\bot^\natural, \mathcal{M}\right) = \bot^\natural \\
\mathsf{APP}^\natural(\mathcal{M}_1, \mathcal{M}_2) &= \mathrm{map}_2\,(\lambda(\mathcal{R}_1,\mathcal{R}_2).\,\mathsf{APP}_v(\mathcal{R}_1,\mathcal{R}_2))\,\mathcal{M}_1\,\mathcal{M}_2 \text{ if } \mathcal{M}_1 \neq \bot^\natural \text{ and } \mathcal{M}_2 \neq \bot^\natural \\
\mathsf{FUN}^{\natural\mathcal{S}}\left(\mathsf{x}, \bot^\natural, \mathcal{M}\right) &= \{\overline{\mathsf{y} \mapsto \mathsf{FUN}^{\mathsf{R}}_v\,(\bot,\bot,\bot)}^{\mathsf{y}\in\mathcal{S}}, \mathsf{this} \mapsto \mathsf{FUN}^{\mathsf{R}}_v\,(\bot,\bot,\bot)\} \\
\mathsf{FUN}^{\natural\mathcal{S}}\left(\mathsf{x}, \mathcal{M}, \bot^\natural\right) &= \mathrm{map}(\lambda\mathcal{R}.\,\mathsf{FUN}^{\mathsf{R}}_v\,(h(\mathcal{R})(\mathcal{M}),\bot,\bot))\,\mathcal{M} \\
\mathsf{FUN}^{\natural\mathcal{S}}(\mathsf{x}, \mathcal{M}_1, \mathcal{M}_2) &= \mathrm{map}_2(\lambda(\mathcal{R}_1,\mathcal{R}_2).\,\mathsf{FUN}^{\mathsf{R}}_v\,(h(\mathcal{R}_1)(\mathcal{M}_1), \mathcal{M}_2(\mathsf{x}), (\mathcal{R}_1;\mathcal{M}_2(\mathsf{x})) \cap \mathcal{R}_2))\,\mathcal{M}_1\,(\mathcal{M}_2\setminus\mathsf{x}) \\
\text{where } h(\mathcal{R})(\mathcal{M}) &= \textstyle\bigcap_{z\in\mathcal{S}\cup\{\mathsf{this}\}}\{(v_1,v_2) \in \mathcal{V}\times\mathcal{V} \mid (v_1,v_2) \in \mathcal{R} \Rightarrow \forall v \in \mathcal{V}, (v,v_2) \in \mathcal{M}(\mathsf{z})\}
\end{aligned}
$$

Fig. 7. Operations on pointwise relations

$$
\begin{aligned}
\mathsf{APP}_v(\mathcal{R}_1,\mathcal{R}_2) &= \{(v,v') \mid \exists v_1,v_2,(v,v_1) \in \mathcal{R}_1 \wedge (v,v_2) \in \mathcal{R}_2 \wedge v_1\,v_2 \rightsquigarrow^\star v'\} \\
\mathsf{FUN}^{\mathsf{R}}_v\left(\mathcal{R}_{\mathrm{arg}},\mathcal{R}_{\mathrm{inner}},\mathcal{R}_{\mathrm{outer}}\right) &= \left\{(v,v') \;\middle|\; \begin{array}{r}\forall v_1,v_2,v'\,v_1 \rightsquigarrow^\star v_2 \Rightarrow (v,v_1) \in \mathcal{R}_{\mathrm{arg}} \Rightarrow \\ (v_1,v_2) \in \mathcal{R}_{\mathrm{inner}} \wedge (v,v_2) \in \mathcal{R}_{\mathrm{outer}}\end{array}\right\} \\
\mathsf{FUN}^{\mathsf{L}}_v\left(\mathcal{R}_{\mathrm{arg}},\mathcal{R}_{\mathrm{inner}},\mathcal{R}_{\mathrm{outer}}\right) &= \left\{(v',v) \;\middle|\; \begin{array}{r}\forall v_1,v_2,v'\,v_1 \rightsquigarrow^\star v_2 \Rightarrow (v_1,v) \in \mathcal{R}_{\mathrm{arg}} \Rightarrow \\ (v_1,v_2) \in \mathcal{R}_{\mathrm{inner}} \wedge (v_2,v) \in \mathcal{R}_{\mathrm{outer}}\end{array}\right\}
\end{aligned}
$$

Fig. 8. Combinators for binary relations on values

The concretisation of correlations (Fig. 9) interprets correlations as binary relations on values. The definition recurses on the syntax of correlations, and interprets each construct with its semantic counterpart. It is a *typed* concretisation: the relation $\gamma^\sharp_{\tau_1\otimes\tau_2}(C)$ relates a value v_1 of type τ_1 to a value v_2 of type τ_2. In this paper, we only consider simple, non-recursive types:

$$\tau \quad ::= \quad \mathsf{unit} \quad | \quad \tau \times \tau \quad | \quad \tau + \tau \quad | \quad \tau \to \tau$$

$$
\begin{aligned}
\gamma^\sharp_{\tau_1\otimes\tau_2}(\bot^\sharp) &= \bot \qquad \gamma^\sharp_{\tau\otimes\tau}(\mathsf{EQ}^\sharp) = \mathsf{EQ} \cap \mathcal{V}_\tau \times \mathcal{V}_\tau \qquad \gamma^\sharp_{\tau_1\otimes\tau_2}(\top^\sharp) = \mathcal{V}_{\tau_1} \times \mathcal{V}_{\tau_2} \\
\gamma^\sharp_{(\tau_1\times\tau_1')\otimes\tau_2}(\mathsf{PAIR}^{\sharp\mathsf{L}}(C,C')) &= \mathsf{PAIR}^{\mathsf{L}}\left(\gamma^\sharp_{\tau_1\otimes\tau_2}(C), \gamma^\sharp_{\tau_1'\otimes\tau_2}(C')\right) \\
\gamma^\sharp_{\tau_1\otimes(\tau_2\times\tau_2')}(\mathsf{PAIR}^{\sharp\mathsf{R}}(C,C')) &= \mathsf{PAIR}^{\mathsf{R}}\left(\gamma^\sharp_{\tau_1\otimes\tau_2}(C), \gamma^\sharp_{\tau_1\otimes\tau_2'}(C')\right) \\
\gamma^\sharp_{(\tau_1+\tau_1')\otimes\tau_2}(\mathsf{SUM}^{\sharp\mathsf{L}}(C,C')) &= \mathsf{SUM}^{\mathsf{L}}\left(\gamma^\sharp_{\tau_1\otimes\tau_2}(C), \gamma^\sharp_{\tau_1'\otimes\tau_2}(C')\right) \\
\gamma^\sharp_{\tau_1\otimes(\tau_2+\tau_2')}(\mathsf{SUM}^{\sharp\mathsf{R}}(C,C')) &= \mathsf{SUM}^{\mathsf{R}}\left(\gamma^\sharp_{\tau_1\otimes\tau_2}(C), \gamma^\sharp_{\tau_1\otimes\tau_2'}(C')\right) \\
\gamma^\sharp_{(\tau_1\to\tau_1')\otimes\tau_2}(\mathsf{FUN}^{\sharp\mathsf{L}}(C_\mathrm{a},C_\mathrm{i},C_\mathrm{o})) &= \mathsf{FUN}^{\mathsf{L}}_v\left(\gamma^\sharp_{\tau_1\otimes\tau_2}(C_\mathrm{a}), \gamma^\sharp_{\tau_1\otimes\tau_1'}(C_\mathrm{i}), \gamma^\sharp_{\tau_1'\otimes\tau_2}(C_\mathrm{o})\right) \cap \mathcal{V}_{\tau_1\to\tau_1'} \times \mathcal{V}_{\tau_2} \\
\gamma^\sharp_{\tau_1\otimes(\tau_2\to\tau_2')}(\mathsf{FUN}^{\sharp\mathsf{R}}(C_\mathrm{a},C_\mathrm{i},C_\mathrm{o})) &= \mathsf{FUN}^{\mathsf{R}}_v\left(\gamma^\sharp_{\tau_1\otimes\tau_2}(C_\mathrm{a}), \gamma^\sharp_{\tau_2\otimes\tau_2'}(C_\mathrm{i}), \gamma^\sharp_{\tau_1\otimes\tau_2'}(C_\mathrm{o})\right) \cap \mathcal{V}_{\tau_1} \times \mathcal{V}_{\tau_2\to\tau_2'} \\
\gamma^\sharp_{\tau_1\otimes\tau_2}(C) &= \bot \text{ in all other cases}
\end{aligned}
$$

Fig. 9. Concretisation of correlations. Subscript "a" stands for "arg", "i" for "inner", and "o" for "outer".

$$
\frac{}{\vdash \bot^\sharp : \tau_1 \otimes \tau_2} \qquad \frac{}{\vdash \mathsf{EQ}^\sharp : \tau \otimes \tau} \qquad \frac{}{\vdash \top^\sharp : \tau_1 \otimes \tau_2} \qquad \frac{\vdash C : \tau_1 \otimes \tau_2 \qquad \vdash C' : \tau_1' \otimes \tau_2}{\vdash \mathsf{PAIR}^{\sharp\mathsf{L}}(C,C') : (\tau_1 \times \tau_1') \otimes \tau_2}
$$

$$
\frac{\vdash C : \tau_1 \otimes \tau_2 \qquad \vdash C' : \tau_1 \otimes \tau_2'}{\vdash \mathsf{PAIR}^{\sharp\mathsf{R}}(C,C') : \tau_1 \otimes (\tau_2 \times \tau_2')} \qquad \frac{\vdash C : \tau_1 \otimes \tau_2 \qquad \vdash C' : \tau_1' \otimes \tau_2}{\vdash \mathsf{SUM}^{\sharp\mathsf{L}}(C,C') : (\tau_1 + \tau_1') \otimes \tau_2} \qquad \frac{\vdash C : \tau_1 \otimes \tau_2 \qquad \vdash C' : \tau_1 \otimes \tau_2'}{\vdash \mathsf{SUM}^{\sharp\mathsf{R}}(C,C') : \tau_1 \otimes (\tau_2 + \tau_2')}
$$

$$
\frac{\vdash C_\mathrm{arg} : \tau_1 \otimes \tau_2 \qquad \vdash C_\mathrm{inner} : \tau_1 \otimes \tau_1' \qquad \vdash C_\mathrm{outer} : \tau_1' \otimes \tau_2}{\vdash \mathsf{FUN}^{\sharp\mathsf{L}}(C_\mathrm{arg}, C_\mathrm{inner}, C_\mathrm{outer}) : (\tau_1 \to \tau_1') \otimes \tau_2} \qquad \frac{\vdash C_\mathrm{arg} : \tau_1 \otimes \tau_2 \qquad \vdash C_\mathrm{inner} : \tau_2 \otimes \tau_2' \qquad \vdash C_\mathrm{outer} : \tau_1 \otimes \tau_2'}{\vdash \mathsf{FUN}^{\sharp\mathsf{R}}(C_\mathrm{arg}, C_\mathrm{inner}, C_\mathrm{outer}) : \tau_1 \otimes (\tau_2 \to \tau_2')}
$$

Fig. 10. Wellformed correlations

We write $\Gamma \vdash t : \tau$ to denote that the program t has type τ in the typing environment Γ. We do not recall the standard definition of the typing judgement for simple types. We write $\mathcal{V}_\tau$ to denote the set of values that have type τ in the empty environment.

We also assign types to correlations. The judgement $\vdash C : \tau_1 \otimes \tau_2$ is defined in Fig. 10 and says that C is a correlation between values of type τ_1 and values of type τ_2. For instance, for function correlations, $\vdash \mathsf{FUN}^{\sharp\mathsf{R}}(C_\mathrm{arg}, C_\mathrm{inner}, C_\mathrm{outer}) : \tau_1 \otimes (\tau_2 \to \tau_2')$ where the correlation on arguments C_arg has type $\tau_1 \otimes \tau_2$ and the correlation on results C_outer has type $\tau_1 \otimes \tau_2'$, whereas the input-output correlation C_inner has type $\tau_2 \otimes \tau_2'$.

We defined in Fig. 11 the inclusion of correlations, written $C_1 \sqsubseteq^\sharp C_2$. It asserts that $\bot^\sharp$ is the bottom element and that $\top^\sharp$ is the top element and relates $\mathsf{EQ}^\sharp$ to itself. The rules for pairs and sums compare elements component-wise, by using a projection operator, similar to that of [Andreescu et al. 2019], that we will discuss later. Finally, function correlations are compared contravariantly for their arguments, and covariantly for their results and for their "inner" correlations.

The projection operator $C \Downarrow_b^S p$ (Fig. 12 and Fig. 13) projects the side S of the correlation on a projection path p. A projection path is either a pair projection path (.1, .2) that projects on one component of a pair, or a sum component (@1, @2) that projects on one case of a sum. The boolean parameter b tells whether the projection should be *strong*, *i.e.* whether the projection should try to project below functions. The strong projection is semantically more precise, but hampers the transitivity of the inclusion of correlations. Therefore, we use the weak projection in the definition

$$\frac{}{\bot^\sharp \sqsubseteq^\sharp C} \qquad \frac{}{\mathsf{EQ}^\sharp \sqsubseteq^\sharp \mathsf{EQ}^\sharp} \qquad \frac{}{C \sqsubseteq^\sharp \top^\sharp} \qquad \frac{C \Downarrow^S_{\mathsf{ff}} .1 \sqsubseteq^\sharp C_1 \qquad C \Downarrow^S_{\mathsf{ff}} .2 \sqsubseteq^\sharp C_2}{C \sqsubseteq^\sharp \mathsf{PAIR}^{\sharp S}(C_1, C_2)}$$

$$\frac{C \Downarrow^S_{\mathsf{ff}} @1 \sqsubseteq^\sharp C_1 \qquad C \Downarrow^S_{\mathsf{ff}} @2 \sqsubseteq^\sharp C_2}{C \sqsubseteq^\sharp \mathsf{SUM}^{\sharp S}(C_1, C_2)} \qquad \frac{C'_{\mathrm{arg}} \sqsubseteq^\sharp C_{\mathrm{arg}} \qquad C_{\mathrm{inner}} \sqsubseteq^\sharp C'_{\mathrm{inner}} \qquad C_{\mathrm{outer}} \sqsubseteq^\sharp C'_{\mathrm{outer}}}{\mathsf{FUN}^{\sharp S}(C_{\mathrm{arg}}, C_{\mathrm{inner}}, C_{\mathrm{outer}}) \sqsubseteq^\sharp \mathsf{FUN}^{\sharp S}(C'_{\mathrm{arg}}, C'_{\mathrm{inner}}, C'_{\mathrm{outer}})}$$

Fig. 11. Inclusion of correlations

$$\begin{aligned}
\bot^\sharp \Downarrow^S_b .i &= \bot^\sharp \qquad \top^\sharp \Downarrow^S_b .i = \top^\sharp \\
\mathsf{EQ}^\sharp \Downarrow^S_b .1 &= \mathsf{PAIR}^{\sharp S^{-1}}(\mathsf{EQ}^\sharp, \top^\sharp) \qquad \mathsf{EQ}^\sharp \Downarrow^S_b .2 = \mathsf{PAIR}^{\sharp S^{-1}}(\top^\sharp, \mathsf{EQ}^\sharp) \\
\mathsf{PAIR}^{\sharp S'}(C_1, C_2) \Downarrow^S_b .i &= \text{if } S = S' \text{ then } C_i \text{ else } \mathsf{PAIR}^{\sharp S'}(C_1 \Downarrow^S_b .i, C_2 \Downarrow^S_b .i) \\
\mathsf{SUM}^{\sharp S'}(C_1, C_2) \Downarrow^S_b .i &= \text{if } S = S' \text{ then } \top^\sharp \text{ else } \mathsf{SUM}^{\sharp S'}(C_1 \Downarrow^S_b .i, C_2 \Downarrow^S_b .i) \\
\mathsf{FUN}^{\sharp S'}(C_{\mathrm{arg}}, C_{\mathrm{inner}}, C_{\mathrm{outer}}) \Downarrow^S_b .i &= \begin{cases} \top^\sharp & \text{if } S = S' \text{ or } b = \mathsf{ff} \text{ or } C_{\mathrm{arg}} \neq \top^\sharp \\ \mathsf{FUN}^{\sharp S'}(C_{\mathrm{arg}} \Downarrow^S_b .i, C_{\mathrm{inner}}, C_{\mathrm{outer}} \Downarrow^S_b .i) & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 12. Projection of correlation on a pair component, where $\mathsf{L}^{-1} = \mathsf{R}$ and $\mathsf{R}^{-1} = \mathsf{L}$.

$$\begin{aligned}
\bot^\sharp \Downarrow^S_b @i &= \bot^\sharp \qquad \top^\sharp \Downarrow^S_b @i = \top^\sharp \\
\mathsf{EQ}^\sharp \Downarrow^S_b @1 &= \mathsf{SUM}^{\sharp S^{-1}}(\mathsf{EQ}^\sharp, \bot^\sharp) \qquad \mathsf{EQ}^\sharp \Downarrow^S_b @2 = \mathsf{SUM}^{\sharp S^{-1}}(\bot^\sharp, \mathsf{EQ}^\sharp) \\
\mathsf{PAIR}^{\sharp S'}(C_1, C_2) \Downarrow^S_b @i &= \text{if } S = S' \text{ then } \top^\sharp \text{ else } \mathsf{PAIR}^{\sharp S'}(C_1 \Downarrow^S_b @i, C_2 \Downarrow^S_b @i) \\
\mathsf{SUM}^{\sharp S'}(C_1, C_2) \Downarrow^S_b @i &= \text{if } S = S' \text{ then } C_i \text{ else } \mathsf{SUM}^{\sharp S'}(C_1 \Downarrow^S_b @i, C_2 \Downarrow^S_b @i) \\
\mathsf{FUN}^{\sharp S'}(C_{\mathrm{arg}}, C_{\mathrm{inner}}, C_{\mathrm{outer}}) \Downarrow^S_b @i &= \begin{cases} \top^\sharp & \text{if } S = S' \text{ or } b = \mathsf{ff} \text{ or } C_{\mathrm{arg}} \neq \top^\sharp \\ \mathsf{FUN}^{\sharp S'}(C_{\mathrm{arg}} \Downarrow^S_b @i, C_{\mathrm{inner}}, C_{\mathrm{outer}} \Downarrow^S_b @i) & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 13. Projection of correlation on a sum case

of the inclusion (to ensure its transitivity), and the strong projection in the other operations (union, intersection, *etc.*).

The transitivity of the ordering relies of the following property: projecting first on side L and then on side R should give the same result as performing the projections on R first and then on L.

Lemma 5.5 (Swap). *Let $p, p' \in \{.1, .2, @1, @2\}$. If $S \neq S'$, then $C \Downarrow^{S'}_{\mathsf{ff}} p' \Downarrow^S_{\mathsf{ff}} p = C \Downarrow^S_{\mathsf{ff}} p \Downarrow^{S'}_{\mathsf{ff}} p'$.*

This property ensures that projection is monotonic with respect to the inclusion.

Lemma 5.6 (Monotonicity of projection). *Let $p \in \{.i, @j\}$. If $C_1 \sqsubseteq^\sharp C_2$, then $C_1 \Downarrow^S_{\mathsf{ff}} p \sqsubseteq^\sharp C_2 \Downarrow^S_{\mathsf{ff}} p$.*

The monotonicity, in turn, is the key property that ensures the transitivity of inclusion.

Lemma 5.7. *Assume $\vdash C_i : \tau \otimes \tau'$ for $i \in \{1, 2, 3\}$. If $C_1 \sqsubseteq^\sharp C_2$ and $C_2 \sqsubseteq^\sharp C_3$, then $C_1 \sqsubseteq^\sharp C_3$.*

The semantic *meaning* of projection is a projection of the values on one side of the relation. For example, the projection $C \Downarrow^{\mathsf{L}}_b .1$ over-approximates the operation $\mathsf{PAIR}^{\mathsf{R}}(\mathsf{EQ}, \top_{\mathsf{v}}) ; \gamma^\sharp_{(\tau_1 \times \tau_2) \otimes \tau'}(C)$. This operation composes on the left a relation that effectively take out the first component of pair values. Similarly, the projection $C \Downarrow^{\mathsf{R}}_b @2$ over-approximates the operation $\gamma^\sharp_{(\tau_1 + \tau_2) \otimes \tau'}(C); \mathsf{SUM}^{\mathsf{L}}(\bot, \mathsf{EQ})$, that composes on the right a relation that retrieves the second case of a sum value.

The soundness results about projection entail the correctness of the inclusion on correlations.

$$
\begin{aligned}
C \sqcup^\sharp \top^\sharp &= \top^\sharp \qquad C \sqcup^\sharp \bot^\sharp = C \qquad \mathsf{EQ}^\sharp \sqcup^\sharp \mathsf{EQ}^\sharp = \mathsf{EQ}^\sharp \\
C \sqcup^\sharp \mathsf{PAIR}^{\sharp S}(C_1, C_2) &= \mathsf{PAIR}^{\sharp S}(C \Downarrow^S_{\mathrm{tt}} .1 \sqcup^\sharp C_1, C \Downarrow^S_{\mathrm{tt}} .2 \sqcup^\sharp C_2) \\
C \sqcup^\sharp \mathsf{SUM}^{\sharp S}(C_1, C_2) &= \mathsf{SUM}^{\sharp S}(C \Downarrow^S_{\mathrm{tt}} @1 \sqcup^\sharp C_1, C \Downarrow^S_{\mathrm{tt}} @2 \sqcup^\sharp C_2) \\
\mathsf{EQ}^\sharp \sqcup^\sharp \mathsf{FUN}^{\sharp S}(C_1, C_2, C_3) &= \top^\sharp \\
\mathsf{FUN}^{\sharp S}(C_1, C_2, C_3) \sqcup^\sharp \mathsf{FUN}^{\sharp S'}(C'_1, C'_2, C'_3) &= \top^\sharp \quad \text{if } S \neq S' \vee \neg(C_1 \sqsubseteq^\sharp C'_1 \vee C'_1 \sqsubseteq^\sharp C_1) \\
\mathsf{FUN}^{\sharp S}(C_1, C_2, C_3) \sqcup^\sharp \mathsf{FUN}^{\sharp S}(C'_1, C'_2, C'_3) &= \mathsf{FUN}^{\sharp S}(C_1, C_2 \sqcup^\sharp C'_2, C_3 \sqcup^\sharp C'_3) \quad \text{if } C_1 \sqsubseteq^\sharp C'_1
\end{aligned}
$$

Fig. 14. Union of correlations (symmetric cases are omitted)

$$
\begin{aligned}
C \sqcap^\sharp \top^\sharp &= C \qquad C \sqcap^\sharp \bot^\sharp = \bot^\sharp \qquad \mathsf{EQ}^\sharp \sqcap^\sharp \mathsf{EQ}^\sharp = \mathsf{EQ}^\sharp \\
C \sqcap^\sharp \mathsf{PAIR}^{\sharp S}(C_1, C_2) &= \mathsf{PAIR}^{\sharp S}(C \Downarrow^S_{\mathrm{tt}} .1 \sqcap^\sharp C_1, C \Downarrow^S_{\mathrm{tt}} .2 \sqcap^\sharp C_2) \\
C \sqcap^\sharp \mathsf{SUM}^{\sharp S}(C_1, C_2) &= \mathsf{SUM}^{\sharp S}(C \Downarrow^S_{\mathrm{tt}} @1 \sqcap^\sharp C_1, C \Downarrow^S_{\mathrm{tt}} @2 \sqcap^\sharp C_2) \\
\mathsf{EQ}^\sharp \sqcap^\sharp \mathsf{FUN}^{\sharp S}(C_1, C_2, C_3) &= \mathsf{FUN}^{\sharp S}(C_1, C_2, C_3) \\
\mathsf{FUN}^{\sharp S}(C_1, C_2, C_3) \sqcap^\sharp \mathsf{FUN}^{\sharp S'}(C'_1, C'_2, C'_3) &= \top^\sharp \quad \text{if } S \neq S' \vee \neg(C_1 \sqsubseteq^\sharp C'_1 \vee C'_1 \sqsubseteq^\sharp C_1) \\
\mathsf{FUN}^{\sharp S}(C_1, C_2, C_3) \sqcap^\sharp \mathsf{FUN}^{\sharp S}(C'_1, C'_2, C'_3) &= \mathsf{FUN}^{\sharp S}(C_1, C_2 \sqcap^\sharp C'_2, C_3 \sqcap^\sharp C'_3) \quad \text{if } C_1 \sqsubseteq^\sharp C'_1
\end{aligned}
$$

Fig. 15. Intersection of correlations (symmetric cases are omitted)

Lemma 5.8. *Assume* $\vdash C_1 : \tau \otimes \tau'$ *and* $\vdash C_2 : \tau \otimes \tau'$. *If* $C_1 \sqsubseteq^\sharp C_2$, *then* $\gamma^\sharp_{\tau\otimes\tau'}(C_1) \subseteq \gamma^\sharp_{\tau\otimes\tau'}(C_2)$.

The definitions of the union (Fig. 14) and composition (Fig. 16) of correlations are similar to those of [Andreescu et al. 2019]. The definition of intersection, similar to union, is given in Fig. 15. The new cases involve functions. For example, $\mathsf{FUN}^{\sharp S}(C_1, C_2, C_3) \sqcup^\sharp \mathsf{FUN}^{\sharp S}(C'_1, C'_2, C'_3) = \mathsf{FUN}^{\sharp S}(C_1, C_2 \sqcup^\sharp C'_2, C_3 \sqcup^\sharp C'_3)$ only when $C_1 \sqsubseteq^\sharp C'_1$. Note that we cannot use the intersection $C_1 \sqcap^\sharp C'_1$ for the correlation of the argument. Due to contravariance, we must use an *under*-approximation of relational intersection, but we only have an over-approximation. Hence, we only give a precise result when one of the arguments is included in the other. Otherwise, we default to $\top^\sharp$.

The *application* of two correlations $\mathsf{APP}^\sharp(C_1, C_2)$ is a novel operation. As we discussed in §5.1, the bottom element is absorbant. The interesting case is the application of functions: we have $\mathsf{APP}^\sharp(\mathsf{FUN}^{\sharp\mathrm{R}}(C_1, C_2, C_3), C) = (C \fatsemi^\sharp C_2) \sqcap^\sharp C_3$ when $C \sqsubseteq^\sharp C_1$. We obtain the same expression as the one used in the abstraction of let-bindings in §5.1, when the correlation of the actual argument is smaller than the correlation of the formal argument. Otherwise, we default to $\top^\sharp$.

The operations on correlations are sound abstractions of their relational counterparts:

Lemma 5.9 (Soundness). *The following assertions hold:*

- *If* $\vdash C_1 : \tau \otimes \tau'$ *and* $\vdash C_2 : \tau \otimes \tau'$, *then* $\gamma^\sharp_{\tau\otimes\tau'}(C_1) \cup \gamma^\sharp_{\tau\otimes\tau'}(C_2) \subseteq \gamma^\sharp_{\tau\otimes\tau'}(C_1 \sqcup^\sharp C_2)$
- *If* $\vdash C_1 : \tau \otimes \tau'$ *and* $\vdash C_2 : \tau \otimes \tau'$, *then* $\gamma^\sharp_{\tau\otimes\tau'}(C_1) \cap \gamma^\sharp_{\tau\otimes\tau'}(C_2) \subseteq \gamma^\sharp_{\tau\otimes\tau'}(C_1 \sqcap^\sharp C_2)$
- *If* $\vdash C_1 : \tau \otimes \tau'$ *and* $\vdash C_2 : \tau' \otimes \tau''$, *then* $\gamma^\sharp_{\tau\otimes\tau'}(C_1); \gamma^\sharp_{\tau'\otimes\tau''}(C_2) \subseteq \gamma^\sharp_{\tau\otimes\tau''}(C_1 \fatsemi^\sharp C_2)$
- *If* $\vdash C_1 : \tau \otimes (\tau' \to \tau'')$ *and* $\vdash C_2 : \tau \otimes \tau'$, *then* $\mathsf{APP}_v\left(\gamma^\sharp_{\tau\otimes(\tau'\to\tau'')}(C_1), \gamma^\sharp_{\tau\otimes\tau'}(C_2)\right) \subseteq \gamma^\sharp_{\tau\otimes\tau''}(\mathsf{APP}^\sharp(C_1, C_2))$

As a consequence, all the relations used in the denotation of a term t (Fig. 4) can be approximated by operations on pointwise correlations, provided t is well typed. The restriction to well typed terms is due to the correlation abstraction (§5.2), that can only abstract *typed* values.

5.3 Examples

From the previous abstractions of the denotational semantics, we obtain a sound analyser for simply typed λ-terms. We have chosen not to exploit the Self rule on functions, to keep the analysis

$$
\begin{aligned}
\bot^\sharp \fatsemi^\sharp C = C \fatsemi^\sharp \bot^\sharp = \bot^\sharp \qquad \top^\sharp \fatsemi^\sharp \top^\sharp = \top^\sharp \qquad \mathsf{EQ}^\sharp \fatsemi^\sharp C = C \fatsemi^\sharp \mathsf{EQ}^\sharp = C \\
\mathsf{PAIR}^{\sharp\mathsf{L}}(C_1, C_2) \fatsemi^\sharp C = \mathsf{PAIR}^{\sharp\mathsf{L}}(C_1 \fatsemi^\sharp C, C_2 \fatsemi^\sharp C) \\
C \fatsemi^\sharp \mathsf{PAIR}^{\sharp\mathsf{R}}(C_1, C_2) = \mathsf{PAIR}^{\sharp\mathsf{R}}(C \fatsemi^\sharp C_1, C \fatsemi^\sharp C_2) \\
\mathsf{PAIR}^{\sharp\mathsf{R}}(C_1, C_2) \fatsemi^\sharp \mathsf{PAIR}^{\sharp\mathsf{L}}(C_1', C_2') = (C_1 \fatsemi^\sharp C_1') \sqcap^\sharp (C_2 \fatsemi^\sharp C_2') \\
\mathsf{PAIR}^{\sharp\mathsf{R}}(C_1, C_2) \fatsemi^\sharp \top^\sharp = (C_1 \fatsemi^\sharp \top^\sharp) \sqcap^\sharp (C_2 \fatsemi^\sharp \top^\sharp) \\
\top^\sharp \fatsemi^\sharp \mathsf{PAIR}^{\sharp\mathsf{L}}(C_1', C_2') = (\top^\sharp \fatsemi^\sharp C_1') \sqcap^\sharp (\top^\sharp \fatsemi^\sharp C_2') \\
\mathsf{SUM}^{\sharp\mathsf{L}}(C_1, C_2) \fatsemi^\sharp C = \mathsf{SUM}^{\sharp\mathsf{L}}(C_1 \fatsemi^\sharp C, C_2 \fatsemi^\sharp C) \\
C \fatsemi^\sharp \mathsf{SUM}^{\sharp\mathsf{R}}(C_1, C_2) = \mathsf{SUM}^{\sharp\mathsf{R}}(C \fatsemi^\sharp C_1, C \fatsemi^\sharp C_2) \\
\mathsf{SUM}^{\sharp\mathsf{R}}(C_1, C_2) \fatsemi^\sharp \mathsf{SUM}^{\sharp\mathsf{L}}(C_1', C_2') = (C_1 \fatsemi^\sharp C_1') \sqcup^\sharp (C_2 \fatsemi^\sharp C_2') \\
\mathsf{SUM}^{\sharp\mathsf{R}}(C_1, C_2) \fatsemi^\sharp \top^\sharp = (C_1 \fatsemi^\sharp \top^\sharp) \sqcup^\sharp (C_2 \fatsemi^\sharp \top^\sharp) \\
\top^\sharp \fatsemi^\sharp \mathsf{SUM}^{\sharp\mathsf{L}}(C_1', C_2') = (\top^\sharp \fatsemi^\sharp C_1') \sqcup^\sharp (\top^\sharp \fatsemi^\sharp C_2') \\
\mathsf{FUN}^{\sharp\mathsf{L}}(C_1, C_2, C_3) \fatsemi^\sharp C = \begin{cases} \mathsf{FUN}^{\sharp\mathsf{L}}(C_1 \fatsemi^\sharp C, C_2, C_3 \fatsemi^\sharp C) & \text{if } C_1 \fatsemi^\sharp C \fatsemi^\sharp C^{-1} \sqsubseteq^\sharp C_1 \\ \top^\sharp & \text{otherwise} \end{cases} \\
C \fatsemi^\sharp \mathsf{FUN}^{\sharp\mathsf{R}}(C_1, C_2, C_3) = \begin{cases} \mathsf{FUN}^{\sharp\mathsf{R}}(C \fatsemi^\sharp C_1, C_2, C \fatsemi^\sharp C_3) & \text{if } C^{-1} \fatsemi^\sharp C \fatsemi^\sharp C_1 \sqsubseteq^\sharp C_1 \\ \top^\sharp & \text{otherwise} \end{cases} \\
\mathsf{FUN}^{\sharp\mathsf{R}}(C_1, C_2, C_3) \fatsemi^\sharp \mathsf{FUN}^{\sharp\mathsf{L}}(C_1', C_2', C_3') = \top^\sharp \\
\mathsf{FUN}^{\sharp\mathsf{R}}(C_1, C_2, C_3) \fatsemi^\sharp \top^\sharp = \top^\sharp \fatsemi^\sharp \mathsf{FUN}^{\sharp\mathsf{R}}(C_1, C_2, C_3) = \top^\sharp
\end{aligned}
$$

Fig. 16. Composition of correlations

$$
\begin{aligned}
\mathsf{APP}^\sharp(\top^\sharp, C) = \mathsf{APP}^\sharp(\mathsf{EQ}^\sharp, C) = \top^\sharp \qquad \mathsf{APP}^\sharp(C, \bot^\sharp) = \mathsf{APP}^\sharp(\bot^\sharp, C) = \bot^\sharp \\
\mathsf{APP}^\sharp(\mathsf{PAIR}^{\sharp\mathsf{L}}(C_1, C_2), C) = \mathsf{PAIR}^{\sharp\mathsf{L}}(\mathsf{APP}^\sharp(C_1, C \Downarrow^{\mathsf{L}}_{\mathsf{tt}} .1), \mathsf{APP}^\sharp(C_2, C \Downarrow^{\mathsf{L}}_{\mathsf{tt}} .2)) \\
\mathsf{APP}^\sharp(\mathsf{PAIR}^{\sharp\mathsf{R}}(C_1, C_2), C) = \bot^\sharp \\
\mathsf{APP}^\sharp(C, \mathsf{PAIR}^{\sharp\mathsf{L}}(C_1, C_2)) = \mathsf{PAIR}^{\sharp\mathsf{L}}(\mathsf{APP}^\sharp(C \Downarrow^{\mathsf{L}}_{\mathsf{tt}} .1, C_1), \mathsf{APP}^\sharp(C \Downarrow^{\mathsf{L}}_{\mathsf{tt}} .2, C_2)) \\
\mathsf{APP}^\sharp(\mathsf{SUM}^{\sharp\mathsf{L}}(C_1, C_2), C) = \mathsf{SUM}^{\sharp\mathsf{L}}(\mathsf{APP}^\sharp(C_1, C \Downarrow^{\mathsf{L}}_{\mathsf{tt}} @1), \mathsf{APP}^\sharp(C_2, C \Downarrow^{\mathsf{L}}_{\mathsf{tt}} @2)) \\
\mathsf{APP}^\sharp(\mathsf{SUM}^{\sharp\mathsf{R}}(C_1, C_2), C) = \bot^\sharp \\
\mathsf{APP}^\sharp(C, \mathsf{SUM}^{\sharp\mathsf{L}}(C_1, C_2)) = \mathsf{SUM}^{\sharp\mathsf{L}}(\mathsf{APP}^\sharp(C \Downarrow^{\mathsf{L}}_{\mathsf{tt}} @1, C_1), \mathsf{APP}^\sharp(C \Downarrow^{\mathsf{L}}_{\mathsf{tt}} @2, C_2)) \\
\mathsf{APP}^\sharp(\mathsf{FUN}^{\sharp\mathsf{L}}(C_1, C_2, C_3), C) = \top^\sharp \\
\mathsf{APP}^\sharp(\mathsf{FUN}^{\sharp\mathsf{R}}(C_1, C_2, C_3), C) = \text{if } C \sqsubseteq^\sharp C_1 \text{ then } (C \fatsemi^\sharp C_2) \sqcap^\sharp C_3 \text{ else } \top^\sharp
\end{aligned}
$$

Fig. 17. Application of correlations

$$
\begin{aligned}
\bot^{\sharp -1} = \bot^\sharp \qquad \mathsf{EQ}^{\sharp -1} = \mathsf{EQ}^\sharp \qquad \top^{\sharp -1} = \top^\sharp \\
\mathsf{PAIR}^{\sharp S}(C_1, C_2)^{-1} = \mathsf{PAIR}^{\sharp(S^{-1})}({C_1}^{-1}, {C_2}^{-1}) \\
\mathsf{SUM}^{\sharp S}(C_1, C_2)^{-1} = \mathsf{SUM}^{\sharp(S^{-1})}({C_1}^{-1}, {C_2}^{-1}) \\
\mathsf{FUN}^{\sharp S}(C_1, C_2, C_3)^{-1} = \mathsf{FUN}^{\sharp(S^{-1})}({C_1}^{-1}, C_2, {C_3}^{-1})
\end{aligned}
$$

Fig. 18. Reverse correlations, where $\mathsf{L}^{-1} = \mathsf{R}$ and $\mathsf{R}^{-1} = \mathsf{L}$.

modular. Using this rule would give a *whole program* analysis. The following analysis results are produced by a prototype implementation in OCaml of the analysis.

Because closed programs are analysed in the empty context, the analyser will return a map of the form $\{\mathsf{this} \mapsto C\}$. In the remainder of the section, we will only show the correlation C.

The analysis infers for the identity function $\mathsf{id} = \lambda \mathsf{x}.\, \mathsf{x}$ the correlation $\mathsf{FUN}^{\sharp R}(\top^\sharp, \mathsf{EQ}^\sharp, \top^\sharp)$. It denotes a function, that requires nothing about its argument, and that ensures nothing about its result, but whose input-output behaviour of the function is the identity. This is the most precise result one could hope for: in this case, the analysis is *complete*.

For the first boolean $\mathsf{K} = \lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathsf{x}$, we get the correlation $\mathsf{FUN}^{\sharp R}(\top^\sharp, C_{\text{inner}}, \top^\sharp)$, where $C_{\text{inner}} = \mathsf{FUN}^{\sharp R}(\top^\sharp, \top^\sharp, \mathsf{EQ}^\sharp)$. This more involved correlation denotes a function whose input-output relation relates any input v with a function that always returns the value v. This is again a complete analysis. As expected, the analysis of $\mathsf{K}\,\mathsf{id}\,\mathsf{K}$ gives the same result as for the identity.

For $\lambda \mathsf{x}.\ (\mathsf{x}, \mathsf{x})$, we obtain the correlation $\mathsf{FUN}^{\sharp R}(\top^\sharp, \mathsf{PAIR}^{\sharp R}(\mathsf{EQ}^\sharp, \mathsf{EQ}^\sharp), \top^\sharp)$, that denotes a function whose input-output behaviour maps any value v to the pair (v, v)—a complete analysis again.

The function $\mathsf{swap} = \lambda \mathsf{x}.\, \lambda \mathsf{y}.\ (\mathsf{y}, \mathsf{x})$ has the correlation $\mathsf{FUN}^{\sharp R}(\top^\sharp, C_{\text{inner}}, C_{\text{outer}})$ where $C_{\text{inner}} = \mathsf{FUN}^{\sharp R}(\top^\sharp, \mathsf{PAIR}^{\sharp R}(\mathsf{EQ}^\sharp, \top^\sharp), \mathsf{PAIR}^{\sharp R}(\top^\sharp, \mathsf{EQ}^\sharp))$ and $C_{\text{outer}} = \mathsf{FUN}^{\sharp R}(\top^\sharp, \mathsf{PAIR}^{\sharp R}(\mathsf{EQ}^\sharp, \top^\sharp), \top^\sharp)$. In other words, we have a function that returns a function. The inner function always returns a pair whose second component is the outer argument. Moreover, the input-output behaviour of the inner function maps any value to a pair whose first component is that value. To summarise, the outer argument is put in the second component of the pair, while the inner argument is put in the first argument of the pair. The analysis computes, again, the most precise result for the extensional behaviour of swap. The analysis thus precisely handles *curried* functions.

The function $\mathsf{unwrap} = \lambda \mathsf{x}.\, \mathsf{match}\ \mathsf{x}\ \mathsf{with}\ \mathsf{inj}_1\, \mathsf{y}_1 \to \mathsf{y}_1 \mid \mathsf{inj}_2\, \mathsf{y}_2 \to \mathsf{y}_2$ of type $(\tau + \tau) \to \tau$ removes the sum injection that wraps a value. It is given the correlation $\mathsf{FUN}^{\sharp R}(\top^\sharp, \mathsf{SUM}^{\sharp L}(\mathsf{EQ}^\sharp, \mathsf{EQ}^\sharp), \top^\sharp)$.

The function $\mathsf{apply} = \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\,\mathsf{x}$, however, is given the correlation $\mathsf{FUN}^{\sharp R}(\top^\sharp, \top^\sharp, \top^\sharp)$: the only inferred information is that it is a function. Indeed, the fact that the result of the function is the result of the application of its two arguments cannot result from the independent attribute abstraction: this relation would relate *several inputs* to the output. The loss of precision comes from the independent attribute abstraction, that degrades the analysis of higher-order *uses* of variables. We could obtain better precision for the applications of *unknown* functions by keeping more relational information between inputs. The further exploration of such improvements is left to future work.

Conventional *whole program* control flow analyses would not give any result for any of the above examples. They would indeed wait for the programs to be further applied to arguments, for them to be precisely analysed. Although our analysis fails at inferring a precise result for apply, it produces precise results for the other examples.

Although our analysis was meant to be a simple example, it is already surprisingly precise, and can favorably compare to standard CFAs, as shown by the following examples.

To illustrate the limitations of 0CFA [Shivers 1991] and justify the need for more precise analyses, Earl et al. [2010] consider the term

$$\begin{array}{l} \mathsf{let}\ \mathsf{f} = \lambda \mathsf{x}.\, \mathsf{x}\ \mathsf{in} \\ \mathsf{let}\ \mathsf{y}_1 = \mathsf{f}\,(\mathsf{inj}_1\,())\ \mathsf{in} \\ \mathsf{let}\ \mathsf{y}_2 = \mathsf{f}\,(\mathsf{inj}_2\,())\ \mathsf{in} \\ \mathsf{y}_1 \end{array}$$

that evaluates to $\mathsf{inj}_1\,()$. Because it does not track context information, 0CFA merges the possible values for x, and infers that the result y_1 can be either $\mathsf{inj}_1\,()$ or $\mathsf{inj}_2\,()$. In contrast, our analyser infers that f is bound to a function that is equivalent to the identity, and this summary is used independently for y_1 and y_2. Our analyser predicts that the result y_1 must necessarily be of the form $\mathsf{inj}_1\,()$. By computing a generic summary for f, and keeping its two instances on $\mathsf{inj}_1\,()$ and $\mathsf{inj}_2\,()$ distinct, our analysis implements a form of polyvariance.

Our analysis also features—by design—a form of data sensitivity. The following program is inpired from the Example 3.27 page 183 of Nielson et al. [1999]:

$$\begin{array}{l}\mathsf{let\ f} = \lambda \mathsf{x}.\ \mathsf{match\ x\ with\ inj}_1\ \mathsf{y}_1 \to \lambda \mathsf{z}_1.\ \mathsf{z}_1 \mid \mathsf{inj}_2\ \mathsf{y}_2 \to \lambda \mathsf{z}_2.\ \mathsf{inj}_1\ () \\ \mathsf{in\ f}\ (\mathsf{inj}_1\ ())\ (\mathsf{inj}_2\ ())\end{array}$$

This program always returns $\mathsf{inj}_2\ ()$. Our modular analyser ensures that the result *must* be of the form $\mathsf{inj}_2\ ()$. This result is on par with what 0CFA would infer. In contrast to 0CFA, the function f is analysed only once and independently of the knowledge of its calling environment, and yet the analysis result remains precise. This is possible, because the summary of f internally contains a case analysis on its argument, due to the pattern matching that occurs in its body.

Finally, the term $\mathsf{let\ f} = \mu \mathsf{f}.\ \lambda \mathsf{x}.\ \mathsf{f}\ (\lambda \mathsf{y}.\ \mathsf{y})\ \mathsf{in\ f}\ \lambda \mathsf{z}.\ \mathsf{z}$ (the "loop" program in Nielson et al. [1999] page 143) is a divergent program. Our analyser accurately infers the $\bot^\sharp$ relation.

6 FORMALISATION IN COQ AND IMPLEMENTATION

We formalised all the results from the previous sections in Coq, except those related to the extension with fixpoints (§4.4). The formalisation notably includes the soundness and completeness results of §4.3, and the soundness of the abstractions of §5 for pointwise relations, and for correlations.

We used the *locally nameless* representation of binders [Charguéraud 2011] to describe the λ-terms, using the Metalib library [Aydemir et al. 2008]. The locally nameless representation is known to require a large amount of infrastructure but has the advantage of providing α-equivalence for free. In this way, the resulting development remains close to a pencil and paper formalisation. Moreover, most of the infrastructure was automatically generated. The Coq development is about 20 kLoC long (excluding infrastructure lemmas) and is available as an accompanying artefact.

Dealing with transpositions and supporting sets for relations (§3) required *a lot* of effort—about 20% of the development. Without the mechanisation effort, we would probably have overlooked the issue of the supporting sets of our stable relations. We also think that the mechanisation process was *crucial* to find the right definitions of §2.

We leave the mechanisation of the fixpoint extension (§4.4) for future work. Indeed, to properly use a least fixed point to interpret recursive functions, one needs to show—while defining the interpretation function—that this very interpretation function is monotone. This is technically challenging in a system like Coq.

We implemented a prototype analyser in OCaml for simply-typed λ-terms, that we used to produce the examples of §5.3. The implementation closely follows the formal definitions of the paper. It also comprises the extension for recursive functions, that performs a Kleene iteration of an abstraction of the functional from §4.4, until it reaches a post-fixpoint. The analysis employs a naive widening operator—not described in this article—that computes an upper bound, and performs no extrapolation. Termination is ensured by a property of the abstract domain implied by simple types: there is a finite number of equivalence classes of correlations of a given type. Supporting recursive types would require more clever widening strategies, which we leave for future work.

7 RELATED WORK

Many researchers have developed analysis techniques for higher-order languages. An exhaustive study of the literature on this topic would go beyond the scope of this paper. We only selected a few previous works, that we think are the most related to this article.

Our notion of stability, and the quantification over larger substitutions in the FUN combinator are related to Kripke semantics, where it is called *persistence* [Kripke 1965; van Benthem 2008].

In the context of strictness analysis of higher-order languages, Cousot and Cousot [1994] introduced *comportment analysis*, an analysis based on abstract interpretation, that generalises existing

work on strictness. They define a collecting semantics for a simply typed λ-calculus, where the meaning of a term is a set of functions from environments to values in some domain. They obtain a comportment semantics, whose definition for variables resembles our SELF relation. Cousot [1997] also uses the same collecting semantics to derive type-based analyses for a call-by-value untyped λ-calculus. A common goal to both works is to abstract *sets of values*. By contrast, we have chosen to abstract relations between input values and output values. The GATHER relation, and the stability condition are specific to our relation-based approach, and have no counterpart in those two articles.

Minimal function graphs (MFG) [Jones and Mycroft 1986] were introduced to interpret programs written in a first-order functional language. MFG really are relations that denote the input-output behaviour of programs. The paper gives an example of constant propagation in that context.

Hudak and Young [1991] define strictness analyses for a non-strict higher-order functional language by abstracting its "collecting interpretation" semantics, that maintains a cache containing the "history" of the evaluation of subterms. When they discuss further work, they affirm: "It is possible to collect not only all values that a particular expression evaluates to, but also all environments that it was evaluated in. This is a straightforward extension [...]". Our semantics precisely tracks values along with their environments, and uncovered subtle problems, such as stability and name binding issues.

Control flow analyses (CFA) [Midtgaard 2012] detect which closures might be called at a call site, by inferring an over-approximation of the sets of values that a subterm might produce. They often introduce *contours* [Shivers 1991] to identify calling contexts and compute context-wise-refined results. Many CFA are *whole program* analyses, that abstract some environment-based denotational semantics or the states of some environment-based abstract machines. They eventually need to abstract closures (pairs of environments and function code), which requires handling the circularity between closures and environments. This has been solved either by forgetting the environment part of a closure [Heintze 1994; Midtgaard and Jensen 2008, 2009]—which leads to variants of 0CFA—or by creating an indirection through the *naming* of environments [Horn and Might 2010]. At the opposite end of the spectrum, [Banerjee and Jensen 2003] define a *modular* type-based analysis that infers control flow information. The collecting semantics we have defined can undoubtedly be used to derive *relational* control flow analyses. The goal of our paper was *not* to perform control flow analysis, though, and neither was the goal of the static analysis we have derived as an example. Interestingly, our denotation for functions exhibits a notion of closure—with the use of GATHER and SELF—although the operational semantics we started with is not environment-based, but rather substitution-based.

The same rule for the denotation of functions also fosters the development of a *modular* analysis, where the body of a function is analysed *only once* at its definition site. *Function summaries* have indeed lead to the successful design of a variety of static analyses including shape analysis [Das 2000; Illous et al. 2017], numerical analysis [Farzan and Kincaid 2015; Jeannet et al. 2004; Kincaid et al. 2017], and correlation analysis [Andreescu et al. 2019]. The correlation abstract domain represents typed relations between values of algebraic datatypes. It is restricted to the analysis of first-order programs, since the abstract domain cannot handle first class functions. The present work effectively extends this work with support for functions as values. As a side product, we obtained new combinators for functions (FUN_v and APP_v), that were not described in the relators of Bird and de Moor [1996].

In the context of the semi-automated verification of higher-order programs, [Kaki and Jagannathan 2014] developed a technique that combines hand-written relational specifications with SMT solvers. They are able to discharge automatically the proof conditions for the verification of complex shape invariants of functional programs, written in a decidable fragment of first-order logic. They rely on the manual definition of the relations, which permits the expression of fine invariants. The

number and the size of annotations can, however, become a burden for large programs. Liquid types [Rondon et al. 2008; Vazou et al. 2014] also leverage SMT solvers to verify rich subset types for Haskell. They are able to reduce the annotation burden by inferring most of the subset types. More recently, Vazou et al. [2017] achieved completeness by embedding a logical variant of the code of functions into their specifications.

Our program logic (§4.1) shares some aspects with work on *characteristic formulæ* for higher-order languages [Charguéraud 2010; Honda et al. 2006]. Such formulæ encode the most general Hoare triple, and can serve as the basis for program verification. They are proved sound and complete with respect to the operational semantics of programs. The predicate AppReturns $f\ x\ P$ from Charguéraud [2010] on the one hand, that asserts that the application of the function f to an argument x terminates and returns a value satisfying P, and, on the other hand, the *evaluation formula* $e_1 \bullet e_2 = e_3$ in the syntax of assertions of Honda et al. [2006], that denotes that the application of e_1 to e_2 converges to a value e_3, are both reminiscent of our APP combinator, whose definition involves the existence of a normal form for the application of two terms. Moreover, the rule for variables from Honda et al. [2006] asserts that the result of evaluating of x must be equal to the value bound to x. This is close to the meaning of our SELF relation.

The SELF rule in our program logic is also analogous to the *strengthening*—also called *selfification*—that occurs in the theory of ML modules [Dreyer et al. 2003; Leroy 2000]. In the context of modules, this operation is used to account precisely for abstract type components in module signatures, by recording the fact that a module signature S is the signature of some module name X. This strengthening can be modelled using singleton kinds [Stone and Harper 2006], where the kind $\mathbb{S}(\tau)$ denotes the $\beta\eta$-equivalence class of the type τ.

8 CONCLUSIONS AND FUTURE WORK

We have introduced a denotational semantics for the untyped call-by-value λ-calculus, where terms are interpreted as *stable* relations between value substitutions and values. The semantics describes the input-output relation that a program realises. This semantics is proved sound and complete with respect to the operational semantics. It also enjoys an equivalent definition in the style of a program logic.

This denotational semantics is a promising *collecting semantics* for the design of relational static analyses for higher-order languages. Following the methodology of abstract interpretation, we have demonstrated the usefulness of our semantics by deriving a static analyser for the simply typed λ-calculus, which we implemented. The analyser infers equalities between the inputs and the outputs of higher-order programs. An outcome of this exercise of deriving a static analyser is the extension of the correlation abstract domain [Andreescu et al. 2019] with support for functions as values. We have used the *independent attribute* abstraction to design the analyser. This lead to a simple development, but also degraded the precision and the expressiveness of the analysis. As future work, we plan to overcome these limitations by keeping more relations between the inputs of a term, or by introducing a form of *polymorphism* over input relations for functions. We will also investigate adding support for exceptions or lazy evaluation to the semantics and the analysis.

We have mechanically verified our results in Coq. This is a first step towards incorporating static-analysis-based reasoning with general purpose proof assistants, such as Coq, Agda or F*. We think that such an integration would alleviate the verification burden for large programs.

Finally, we think that our collecting semantics could also serve as a basis to design *new* static analyses for higher order programs, that could have the potential to trigger more aggressive optimisations in compiler backends. In particular, exploring further the link with CFA, as well as extending CFA with numeric abstract domains constitute natural research directions.

REFERENCES

Samson Abramsky. 1991. Domain Theory in Logical Form. *Annals of Pure and Applied Logic* 51, 1-2 (March 1991), 1–77. https://doi.org/10.1016/0168-0072(91)90065-t

Oana F. Andreescu, Thomas Jensen, Stéphane Lescuyer, and Benoît Montagu. 2019. Inferring Frame Conditions with Static Correlation Analysis. *Proc. ACM Program. Lang.* 3, POPL, Article 47 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290360

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '08*. ACM Press. https://doi.org/10.1145/1328438.1328443

Anindya Banerjee and Thomas Jensen. 2003. Modular Control-Flow Analysis with Rank 2 Intersection Types. *Mathematical Structures in Computer Science* 13, 1 (Feb. 2003), 87–124. https://doi.org/10.1017/s0960129502003845

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects (LNCS, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387. https://doi.org/10.1007/11804192_17

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2005. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (LNCS, Vol. 3362)*, Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–69. https://doi.org/10.1007/978-3-540-30569-9_3

Richard Bird and Oege de Moor. 1996. The Algebra of Programming. https://doi.org/10.1007/978-3-642-61455-2_12

David Cachera and David Pichardie. 2010. A Certified Denotational Abstract Interpreter. In *Interactive Theorem Proving*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 9–24. https://doi.org/10.1007/978-3-642-14052-5_3

Arthur Charguéraud. 2010. Program Verification through Characteristic Formulae. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 321–332. https://doi.org/10.1145/1863543.1863590

Arthur Charguéraud. 2011. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (May 2011), 363–408. https://doi.org/10.1007/s10817-011-9225-2

Stephen A. Cook. 1978. Soundness and Completeness of an Axiom System for Program Verification. *SIAM J. Comput.* 7, 1 (Feb. 1978), 70–90. https://doi.org/10.1137/0207005

Patrick Cousot. 1997. Types as Abstract Interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '97*. ACM Press. https://doi.org/10.1145/263699.263744

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1994. Higher-Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection and PER Analysis of Functional Languages). In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL94)*. IEEE Comput. Soc. Press. https://doi.org/10.1109/iccl.1994.288389

Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Compiler Construction (LNCS, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–179. https://doi.org/10.1007/3-540-45937-5_13

Manuvir Das. 2000. Unification-Based Pointer Analysis with Directional Assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 35–46. https://doi.org/10.1145/349299.349309

Derek Dreyer, Karl Crary, and Robert Harper. 2003. A Type System for Higher-Order Modules. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. Association for Computing Machinery, New York, NY, USA, 236–249. https://doi.org/10.1145/604131.604151

Christopher Earl, Matthew Might, and David Van Horn. 2010. Pushdown Control-Flow Analysis of Higher-Order Programs. *Workshop on Scheme and Functional Programming* abs/1007.4268 (2010). arXiv:1007.4268 http://arxiv.org/abs/1007.4268

Azadeh Farzan and Zachary Kincaid. 2015. Compositional Recurrence Analysis. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design (FMCAD '15)*. FMCAD Inc, Austin, TX, 57–64. http://dl.acm.org/citation.cfm?id=2893529.2893544

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems (LNCS, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Robert W. Floyd. 1993. Assigning Meanings to Programs. In *Program Verification*. Springer Netherlands, 65–81. https://doi.org/10.1007/978-94-011-1793-7_4

Murdoch Gabbay and Andrew M. Pitts. 1999. A New Approach to Abstract Syntax Involving Binders. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*. IEEE Comput. Soc, 214–224. https://doi.org/10.1109/lics.1999.782617

Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. 2011. CertiKOS: A Certified Kernel for Secure Cloud Computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems - APSys '11*. ACM Press. https://doi.org/10.1145/2103799.2103803

Nevin Heintze. 1994. Set-Based Analysis of ML Programs. *ACM SIGPLAN Lisp Pointers* VII, 3 (July 1994), 306–317. https://doi.org/10.1145/182590.182495

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Kohei Honda, Martin Berger, and Nobuko Yoshida. 2006. Descriptive and Relative Completeness of Logics for Higher-Order Functions. In *Automata, Languages and Programming*, Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 360–371. https://doi.org/10.1007/11787006_31

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming - ICFP '10*. ACM Press. https://doi.org/10.1145/1863543.1863553

Paul Hudak and Jonathan Young. 1991. Collecting Interpretations of Expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (April 1991), 269–290. https://doi.org/10.1145/103135.103139

Hugo Illous, Matthieu Lemerre, and Xavier Rival. 2017. A Relational Shape Abstract Domain. In *NASA Formal Methods (LNCS, Vol. 10227)*, Clark Barrett, Misty Davies, and Temesghen Kahsai (Eds.). Springer International Publishing, 212–229. https://doi.org/10.1007/978-3-319-57288-8_15

Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. 2004. A Relational Approach to Interprocedural Shape Analysis. In *Static Analysis (LNCS, Vol. 3148)*, Roberto Giacobazzi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 246–264. https://doi.org/10.1007/978-3-540-27864-1_19

Neil D. Jones and Steven S. Muchnick. 1980. Complexity of Flow Analysis, Inductive Assertion Synthesis and a Language Due to Dijkstra. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. IEEE. https://doi.org/10.1109/sfcs.1980.16

Neil D. Jones and Alan Mycroft. 1986. Data Flow Analysis of Applicative Programs Using Minimal Function Graphs. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '86*. ACM Press. https://doi.org/10.1145/512644.512672

G. Kahn. 1987. Natural Semantics. In *STACS 87*. Springer-Verlag, 22–39. https://doi.org/10.1007/bfb0039592

Gowtham Kaki and Suresh Jagannathan. 2014. A Relational Framework for Higher-Order Shape Analysis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 311–324. https://doi.org/10.1145/2628136.2628159

Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional Recurrence Analysis Revisited. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 248–262. https://doi.org/10.1145/3062341.3062373

Gerwin Klein, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, and Rafal Kolanski. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*. ACM Press. https://doi.org/10.1145/1629575.1629596

Saul A. Kripke. 1965. Semantical Analysis of Intuitionistic Logic I. In *Formal Systems and Recursive Functions*. Elsevier, 92–130. https://doi.org/10.1016/s0049-237x(08)71685-9

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2000. A Modular Module System. *Journal of Functional Programming* 10, 3 (2000), 269–303. https://doi.org/10.1017/S0956796800003683

Xavier Leroy. 2006. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL'06*. ACM Press. https://doi.org/10.1145/1111037.1111042

Claude Marché and Christine Paulin-Mohring. 2005. Reasoning About Java Programs with Aliasing and Frame Conditions. In *Theorem Proving in Higher Order Logics (LNCS, Vol. 3603)*, Joe Hurd and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 179–194. https://doi.org/10.1007/11541868_12

Bertrand Meyer. 2015. Framing the Frame Problem. In *Dependable Software Systems Engineering*. 193–203. https://doi.org/10.3233/978-1-61499-495-4-193

Jan Midtgaard. 2012. Control-Flow Analysis of Functional Programs. *Comput. Surveys* 44, 3 (June 2012), 1–33. https://doi.org/10.1145/2187671.2187672

Jan Midtgaard and Thomas Jensen. 2008. A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In *Static Analysis*. Springer Berlin Heidelberg, 347–362. https://doi.org/10.1007/978-3-540-69166-2_23

Jan Midtgaard and Thomas P. Jensen. 2009. Control-Flow Analysis of Function Calls and Returns by Abstract Interpretation. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming - ICFP '09*. ACM Press. https://doi.org/10.1145/1596550.1596592

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-03811-6

Andrew Pitts. 2016. Nominal Techniques. *ACM SIGLOG News* 3, 1 (Feb. 2016), 57–72. https://doi.org/10.1145/2893582.2893594

J. C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Comput. Soc. https://doi.org/10.1109/lics.2002.1029817

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

Olin Shivers. 1991. The Semantics of Scheme Control-Flow Analysis. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91)*. Association for Computing Machinery, New York, NY, USA, 190–198. https://doi.org/10.1145/115865.115884

Christopher A. Stone and Robert Harper. 2006. Extensional Equivalence and Singleton Types. *ACM Transactions on Computational Logic* 7, 4 (Oct. 2006), 676–722. https://doi.org/10.1145/1183278.1183281

Christian Urban and Cezary Kaliszyk. 2012. General Bindings and Alpha-Equivalence in Nominal Isabelle. *Logical Methods in Computer Science* Volume 8, Issue 2 (June 2012). https://doi.org/10.2168/LMCS-8(2:14)2012

Johan van Benthem. 2008. The Information in Intuitionistic Logic. *Synthese* 167, 2 (Oct. 2008), 251–270. https://doi.org/10.1007/s11229-008-9408-5

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming - ICFP '14*. ACM Press. https://doi.org/10.1145/2628136.2628161

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141